Published on **XML.com** http://www.xml.com/pub/a/2000/08/holman/index.html
See this if you're having trouble printing code examples

What Is XSLT

By G. Ken Holman
August 16, 2000

Introduction

Now that we are successfully using XML to mark up our information according to our own vocabularies, we are taking control and responsibility for our information, instead of abdicating such control to product vendors. These vendors would rather lock our information into their proprietary schemes to keep us beholden to their solutions and technology.

But the flexibility inherent in the power given to each of us to develop our own vocabularies, and for industry associations, e-commerce consortia, and the W3C to develop their own vocabularies, presents the need to be able to transform information marked up in XML from one vocabulary to another.

Two W3C Recommendations, XSLT (the Extensible Stylesheet Language Transformations) and XPath (the XML Path Language), meet that need. They provide a powerful implementation of a tree-oriented transformation language for transmuting instances of XML using one vocabulary into either simple text, the legacy HTML vocabulary, or XML instances using any other vocabulary imaginable. We use the XSLT language, which itself uses XPath, to specify how an implementation of an XSLT processor is to create our desired output from our given marked-up input.

XSLT enables and empowers interoperability. This XML.com introduction strives to overview essential aspects of understanding the context in which these languages help us meet our transformation requirements, and to introduce substantive concepts and terminology to bolster the information available in the W3C Recommendation documents themselves.

Since April 1999 Crane Softwrights Ltd. has published commercial training material titled Practical Transformation Using XSLT and XPath, covering the entire scope of the W3C XSLT and XPath through working drafts and the final 1.0 recommendations. This material is delivered by Crane in instructor-led sessions and is licensed to other training organizations around the world needing to teach these exciting technologies.

Crane has rewritten the first two chapters of this material into prose. These prose-oriented chapters are published on XML.com correspondingly as two main sections. The material assumes no prior knowledge of XSLT and XPath and guides the reader through background, context, structure, concepts and introductory terminology.

Table of Contents

1. The Context of XSL Transformations and the XML Path Language

This first chapter examines the context of two W3C Recommendations -- Extensible Stylesheet Language Transformations (XSLT) and XML Path Language (XPath) -- within the growing family of Recommendations related to the Extensible Markup Language (XML). Later we will look at detailed examples, but first let's focus on XSLT and XPath in the context of a few of the Recommendations in the XML family and examine how these two Recommendations work together to address separate and distinct functionality required when working with structured information technologies.

This chapter does not attempt to address all of the numerous XML-related Recommendations currently released or in development. Specifically, we will be looking at only the following as they relate to XSLT and XPath:

Extensible Markup Language (XML)

For years, applications and vendors have imposed their constraints on the way we can represent our information. Our data has been created, maintained, stored and archived according to the rules enforced by others. The advent of the Extensible Markup Language (XML) moves the control of our information out of the hands of others and into our *own* by providing two basic facilities.

XML describes rules for structuring our information using embedded markup of our own choice. We can take control of our information representation by creating and using a vocabulary we design of elements and attributes that makes sense for the way we do our business and use our data.

In addition, XML describes a language for formally declaring the vocabularies we use. This allows our tools to constrain the creation of an instance of our information, and allows our users to validate a properly created instance of information against our set of constraints.

Note 1:	An XML *document* is just an instance of well-formed XML. The two terms *document* and *instance* could be used interchangeably, but this reference material uses the term *instance* to help readers remember that XML isn't just for documents or documentation. With XML we describe a related set of information in a tree-like hierarchical fashion, and gain the benefits of having done so, whether the information captures an invoice-related transaction between computers, or the content of a user manual rendered on paper.

XML Path Language (XPath)

XPath is a string syntax for building addresses to the information found in an XML document. We use this language to specify the locations of document structures or data found in an XML document when processing that information using XSLT. XPath allows us from any location to address any other location or content.

Related Reading

Extensible Stylesheet Language Family (XSLT/XSL)

Two vocabularies specified in separate W3C Recommendations provide for the two distinct styling processes of transforming and rendering XML instances.

We can transform information using one vocabulary into an alternate form by using the Extensible Stylesheet

Language Transformations (XSLT).

The Extensible Stylesheet Language (XSL) is a rendering vocabulary describing the semantics of formatting information for different media.

Namespaces

We use XML namespaces to distinguish information when mixing multiple vocabularies in a single instance. Without namespaces our processes would find the information ambiguous when identical names have been chosen by the designers of the vocabularies we use.

Stylesheet Association

We declare our choice of an associated stylesheet for an XML instance by embedding the construct described in the Stylesheet Association Recommendation. Recipients and applications can choose to respect or ignore this choice, but the declaration indicates that we have tied some process (typically rendering) to our data, which specifies how to consume or work with our information.

1.1 The XML family of Recommendations

Now let's look at the objectives of these selected Recommendations.

1.1.1 Extensible Markup Language (XML)

Historically, the ways we have expressed, created, stored and transmitted our electronic information have been constrained and controlled by the vendors we choose and the applications we run. Alternatively, we now can express our data in a structured fashion oriented around our perspective of the nature of the information itself rather than the nature of an application's choice of how to represent our information. With Extensible Markup Language (XML), we describe our information using embedded markup of elements, attributes and other constructs in a tree-like structure.

- `http://www.w3.org/TR/REC-xml`

1.1.1.1 Structuring information

Contrasted to a file format where information identification relies on some proprietary hidden format, predetermined ordering, or some kind of explicit labeling, the tree-like hierarchical storage structure infers relationships by the scope of values encompassing the scopes of other values.

Though trees shape a number of areas of XML, both logically (markup) and physically (entities such as files or other resources), they are not the only means by which relationships are specified. For example, a quantum of information can arbitrarily point or refer to other information elsewhere through use of unique identifiers.

Two basic objectives of representing information hierarchically are satisfied by the XML Recommendation. It provides:

- an unambiguous mechanism for constraining structure in a stream of information

  XML defines the concept of well-formedness. Well-formedness dictates the syntax used for markup languages within the content of an instance of information. This is the syntax of using angle brackets ("<" and ">") and the ampersand ("&") to demarcate and identify constituent components of information within a file, a resource or a bound data stream. Users of the Hypertext Markup Language (HTML) will recognize the use of these characters for marking the vocabulary described by the designers of the World Wide Web in their web documents.

- a language for specifying how a system can constrain the allowed logical hierarchy of information structures

  XML defines the concept of validity with a syntax for a meta-markup language used to specify vocabularies. A Document Type Definition (DTD) describes the structural schema mandating the user-defined constraints on well-formed information. The designers of HTML have formalized their vocabulary through such a DTD, thus declaring the allowed or expected relationships between components of a hypertext document.

There is an implicit document model for an instance of well-formed XML defined by the mere presence of nested elements found in the information. There is no need to declare this model because the syntax rules governing well-formedness guarantee the information to be seen properly as a hierarchy. As with all hierarchies, there are family-tree-like relationships of parent, child, and sibling constructs relative to each construct found.

Consider the following well-formed XML instance `purc.xml`:

```
<?xml version="1.0"?>
<purchase id="p001">
  <customer db="cust123"/>
  <product db="prod345">
    <amount>23.45</amount>
  </product>
</purchase>
```

Example 1-1: A well-formed XML purchase order instance.

Observe the content nesting (whitespace has been added only for illustrative purposes). The instance follows the lexical rules for XML markup and the hierarchical model is implicit by the nesting of elements. Pay particular attention to the markup on line 3 for the empty element named `customer`, with the attribute named `db`. It will be used later in examples throughout this chapter. The customer element is a child of the document element, which is named `purchase`.

Although the presence of an explicit formal document model is useful to an XML processor or to a system working with XML instances, that model has no impact on the implicit structural model and only minor influence on the interpretation of content found in the instance. This point holds true whether the model is expressed in a DTD or in some of the other Recommendations for structural and content schemata being developed.

Consider the following valid XML instance `purcdtd.xml`:

```
01  <?xml version="1.0"?>
02  <!DOCTYPE purchase [
03  <!ELEMENT purchase ( customer, product+ )>
04  <!ATTLIST purchase id ID #REQUIRED>
05  <!ELEMENT customer EMPTY>
06  <!ATTLIST customer db CDATA #REQUIRED>
07  <!ELEMENT product  ( amount )>
08  <!ATTLIST product  db CDATA #REQUIRED>
09  <!ELEMENT amount   ( #PCDATA )>
10  ]>
11  <purchase id="p001">
12    <customer db="cust123"/>
13    <product db="prod345">
14      <amount>23.45</amount>
15    </product>
16  </purchase>
```

Example 1-2: A valid XML purchase order instance

See how the information content is no different from the previous example, but in this case an explicit document model using XML 1.0 DTD syntax is included (it could have been included by reference to a separate resource). A processor can validate that the information content conforms not only to the lexical rules for XML (well-formedness) but also the syntax rules dictated by the supplied document model (validity).

Looking at the same `customer` element as before (now on line 12), the document model indicates on line 6 that the `db` attribute is, indeed, required: if the attribute is absent the XML processor can report syntactic model constraint violation even if the element is otherwise lexically well-formed. The document model can also provide additional information not evident without a document model (such as the information on line 4 that the `id` attribute for `purchase` is of XML type `ID`).

1.1.1.2 No built-in meanings or concepts

The area of semantics associated with XML instances is very gray. A document model is but one component used to help describe the semantics of the information found in an instance. While well-formed instances do not have a formal document model, often the names of the constructs used within the instances give hints to the associated semantics. Without a formalism yet available in our community to express semantics in a rigorous fashion, we users of XML do (or should!) capture the semantics of a given vocabulary in prose, whether or not the document model is formalized.

The XML 1.0 Recommendation only describes the behavior required of an XML processor acting on an XML stream, and how it must identify constituent data and provide that data to an application using the processor:

Since there are no formalized semantic description facilities in XML, any XML that is used is not tied to any one particular concept or application. There are no rendition or transformation rules or constructs defined in XML. The only purpose of XML is to unambiguously identify and deliver constituent components of data. There are no inherent meanings or semantics of any kind associated with element types defined in a document model. There are no defined controls for implying any rendering semantics.

Even the `xml:space` attribute allowing for the differentiation of whitespace found in a document is not an aspect of rendering but of information description. The author or modeler of an instance is indicating with this reserved attribute (termed "special" in XML 1.0) the nature of the information and how the whitespace found in the information is to be either preserved or handled by a processor in a default fashion.

Some new users of XML who have a background in a markup language such as HTML often assume a magical association of semantics with element types of the same names they have been exposed to in their prior work. In a web page, they can safely assume that the construct `<p>` will be interpreted as a paragraph or `<em>` as emphasized text. However, this interpretation is solely the purview of the designers of HTML and user agents attempting to conform to the World Wide Web Consortium (W3C)-published semantics. Nothing is imposed by any process when creating a new XML vocabulary that happens to use the same names. Applications using XML processors to access XML information must be instructed how to interpret and implement the desired semantics.

1.1.2 XML Path Language (XPath)

Assuming that we have structured our information using XML, how are we going to talk about (address) what is inside our documents? Locating information in an XML document is critical to both transforming it and to associating or relating it to other information. When we write stylesheets and use linking languages, we can address components of our information for a processor by our use of the XML Path Language, also called XPath:

- http://www.w3.org/TR/xpath

1.1.2.1 Addressing structured information

The W3C working group responsible for stylesheets collaborated with the W3C working group responsible for the next generation of hyperlinking to produce XPath as a common base for addressing requirements shared by their respective Recommendations. Both groups extend the core XPath facilities to meet the needs they have in each of their domains: the stylesheet group uses XPath as the core of expressions in XSLT; the linking group uses XPath as the core of expressions in the XPointer Recommendation.

In order to address components you have to know the addressing scheme with which the components are arranged. The basis of addressing XML documents is an abstract data model of interlinked nodes arranged hierarchically echoing the tree-shape of the nested elements in an instance. Nodes of different types make up this hierarchy, each node representing the parsed result of a syntactic structure found in the bytes of the XML instance.

This abstraction insulates addressing from the multiple syntactic forms of given XML constructs, allowing us to focus on the information itself and not the syntax used to represent the information.

Note 2: We see XML documents as a stream or string of bytes that follow the rules of the XML 1.0 Recommendation. Stylesheets do not regard instances in this fashion, and we have to change the way we think of our XML documents in order to successfully work with our information. This leap of understanding ranks high on the list of key aspects of stylesheet writing I needed to internalize before successfully using this technology.

We are given tools to work in the framework provided by the abstraction: a set of data types used to represent values found in the generalization, and a set of functions we use to manipulate and examine those values. The data types include strings, numbers, boolean values and sets of nodes of our information. The functions allow us to cast these values into other data type representations and to return massaged information according to our needs.

1.1.2.2 Addressing identifies a hierarchical position or positions

XPath defines common semantics and syntax for addressing XML-expressed information, and bases these primarily on the hierarchical position of components in the tree. This ordering is referred to as document order in XPath, while in other contexts this is often termed either parse order or depth-first order. Alternatively, we can access an arbitrary location in the tree based on points in the tree having unique identifiers.

We convey XPath addresses in a simple and compact non-XML syntax. This allows us to use an XPath expression as the value of an attribute in an XML vocabulary as in the following examples:

```
select="answer"
```

Example 1-3: A simple XPath expression in a `select` attribute

The above attribute value expresses all children named "`answer`" of the current focus element.

```
match="question|answer"
```

Example 1-4: An XPath expression in a `match` attribute

The above attribute value expresses a test of an element being in the union of the element types named "`question`" and "`answer`".

The XPath syntax looks a lot like addressing subdirectories in a file system or as part of a Universal Resource Identifier (URI). Multiple steps in a location path are separated by either one or two oblique "`/`" characters. Filters can be specified to further refine the nature of the components of our information being addressed.

```
select="question[3]/answer[1]"
```

Example 1-5: A multiple step XPath expression in a `select` attribute

The above example selects only the first "`answer`" child of the third "`question`" child of the focus element.

```
select="id('start')//question[@answer='y']"
```

Example 1-6: A more complex XPath expression in a `select` attribute

The above example uses an XPath address identifying some descendants of the element in the instance that has the unique identifier with the value "`start`". Those identified are the question elements whose answer attribute is equal to the string equal to the lower-case letter 'y'. The value returned is the set of nodes representing the elements meeting the conditions expressed by the address. The address is used in a `select` attribute, thus the XSLT processor is selecting all of the addressed elements for some kind of processing.

1.1.2.3 XPath is *not* a query language

It is important to remember that addressing information is only one aspect of querying information. Other aspects include query operators that massage intermediate results into a final result. While a few operators and functions are available in XSLT to use values identified in documents, these are oriented to string processing, not to complex operations required by some applications.

Note 3:	When query Recommendations are developed, I would hope that the addressing portion is based on XPath as a core, just as with XSLT.

This is a prose version of an excerpt from the book Practical Transformation Using XSLT and XPath (Eighth Edition ISBN 1-894049-05-5 at the time of this writing) published by Crane Softwrights Ltd., written by G. Ken Holman; this excerpt was edited by Stan Swaren, and reviewed by Dave Pawson.

1. The Context of XSL Transformations and the XML Path Language (cont'd)

1.1.3 Styling structured information

1.1.3.1 Styling is *transforming* and *formatting* information

Styling is the rendering of information into a form suitable for consumption by a target audience. Because the audience can change for a given set of information, we often need to apply different styling for that information in order to obtain dissimilar renderings in order to meet the needs of each audience. Perhaps some information needs to be rearranged to make more sense for the reader. Perhaps some information needs to be highlighted differently to bring focus to key content.

It is important when we think about styling information to remember that two distinct processes are involved, not just one. First, we must transform the information from the organization used when it was created into the organization needed for consumption. Second, when rendering we must express, whatever the target medium, the aspects of the appearance of the reorganized information.

Consider the flow of information as a streaming process where information is created upstream and processed or consumed downstream. Upstream, in the early stages, we should be expressing the information abstractly, thus preventing any early binding of concrete or final-form concepts. Midstream, or even downstream, we can exploit the information as long as it remains flexible and abstract. Late binding of the information to a final form can be based on the target use of the final product; by delaying this binding until late in the process, we preserve the original information for exploitation for other purposes along the way.

It is a common but misdirected practice to model information based on how you plan to use it downstream. It does not matter if your target is a presentation-oriented structure, for example, or a structure that is appropriate for another markup-based system. Modeling practice should focus on both the business reasons and inherent relationships existing in the semantics behind the information being described (as such the vocabularies are then content-oriented). For example, emphasized text is often confused with a particular format in which it is rendered. Where we could model information using a `<b>` element type for eventual rendering in a bold face, we would be better off modeling the information using an `<emph>` element type. In this way we capture the reason for marking up information (that it is emphasized from surrounding information), and we do not lock the downstream targets into only using a bold face for rendering.

Many times the midstream or downstream processes need only rearrange, re-label or synthesize the information for a target purpose and never apply any semantics of style for rendering purposes. Transformation tasks stand alone in such cases, meeting the processing needs without introducing rendering issues.

One caveat regarding modeling content-oriented information is that there are applications where the content-orientation is, indeed, presentation-oriented. Consider book publishing where the abstract content is based on presentational semantics. This is meaningful because there is no abstraction beyond the appearance or presentation of the content.

Consider the customer information in Example 1-1. A web user agent doesn't know how to render an element named `<customer>`. The HTML vocabulary used to render the customer information could be as follows:

```
<p>From: <i>(Customer Reference) <b>cust123</b></i>
</p>
```

Example 1-7: HTML rendering semantics markup for example

The rendering result would then be as follows, with the rendering user agent interpreting the markup for italics and boldface presentation semantics:

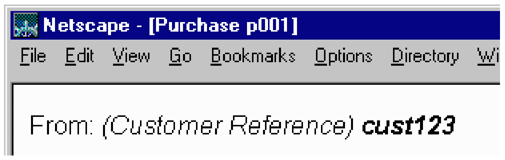

Netscape - [Purchase p001]
File Edit View Go Bookmarks Options Directory Wi
From: *(Customer Reference)* ***cust123***

Figure 1-1: HTML rendering for example

The above illustrates these two distinct styling steps: *transforming* the instance of the XML vocabulary into a new instance according to a vocabulary of rendering semantics; and *formatting* the instance of the rendering vocabulary in the user agent.

1.1.3.2 Two W3C Recommendations

In order to meet these two distinct processes in a detached (yet related) fashion, the W3C Working Group responsible for the Extensible Stylesheet Language (XSL) split the original drafts of their work into two separate Recommendations: one for transforming information and the other for rendering information.

The XSL Transformations (XSLT) 1.0 Recommendation describes a vocabulary recognized by an XSLT processor to transform information from an organization in the source file into a different organization suitable for continued downstream processing.

The Extensible Stylesheet Language (XSL) Working Draft describes a vocabulary recognized by a rendering agent to reify abstract expressions of format into a particular medium of presentation.

Both XSLT and XSL are endorsed by members of WSSSL, an association of researchers and developers passionate about the application of markup technologies in today's information technology infrastructure.

1.1.4 Extensible Stylesheet Language (XSL)

When we need to present our structured information in a given medium or different media, we all have common needs for how the result appears and way the result flows through that appearance. The XSL Working Draft describes the current work developing a vocabulary of formatting and flow semantics that can be expressed using an XML model of elements and attributes:

- `http://www.w3.org/TR/WD-xsl`

1.1.4.1 Formatting and flow semantics vocabulary

This hierarchical vocabulary captures formatting semantics for rendering textual and graphic information in different media. A rendering agent is responsible for interpreting an instance of the vocabulary for a given medium to reify a final result.

This is no different in concept and architecture than using HTML and Cascading Stylesheets (CSS) as a hierarchical vocabulary for rendering a set of information in a web browser. In essence, we are transforming our XML documents into their final display form by transforming instances of our XML vocabularies into instances of a particular rendering vocabulary.

This Working Draft normatively references XSLT as an integral component of XSL. A stylesheet could be written with both the transformation vocabulary and the formatting semantics vocabulary together; it would style an XML instance by rendering the results of transformation. This result need not be serialized in XML syntax; rather, an XSLT/XSL processor can utilize the result of transformation to create a rendered result by interpreting the abstract hierarchy of information without seeing syntax.

1.1.4.2 Target of transformation

When using a formatting semantics vocabulary as the rendering language, the objective for a stylesheet writer is to convert an XML instance of some arbitrary XML vocabulary into an instance of the formatting semantics vocabulary. The result of transformation cannot contain any user-defined vocabulary construct (for example, an address, customer identifier, or purchase order number construct) because the rendering agent would not know what to do with constructs labeled with these foreign, unknown identifiers.

Consider two examples: HTML for rendering in a web browser and XSL for rendering on screen, on paper or audibly. In both cases, the rendering agents only understand the vocabulary expressing their respective formatting semantics and wouldn't know what to do with alien element types defined by the user.

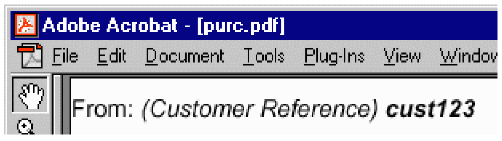
Just as with HTML, a stylesheet writer utilizing XSL for rendering must transform each and every user construct into a rendering construct to direct the rendering agent to produce the desired result. By learning and understanding the semantics behind the constructs of XSL formatting, the stylesheet writer can create an instance of the formatting vocabulary expressing the desired layout of the final result (e.g. area geometry, spacing, font metrics, etc.), with each piece of information in the result coming from either the source data or the stylesheet itself.

Consider once more the customer information in Example 1-1. An XSL rendering agent doesn't know how to render a marked up construct named `<customer>`. The XSL vocabulary used to render the customer information could be as follows:

```
<fo:block space-before.optimum="20pt" font-size="20pt">From:
<fo:inline-sequence font-style="italic">(Customer Reference)
<fo:inline-sequence font-weight="bold">cust123</fo:inline-sequence>
</fo:inline-sequence>
</fo:block>
```

Example 1-8: XSL rendering semantics markup for example

The rendering result when using the Portable Document Format (PDF) would then be as follows, with an intermediate PDF generation step interpreting the XSL markup for italics and boldface presentation semantics:


Adobe Acrobat - [purc.pdf]
File Edit Document Tools Plug-Ins View Windov

From: *(Customer Reference)* ***cust123***

Figure 1-2: XSL rendering for example

The above again illustrates the two distinctive styling steps: *transforming* the instance of the XML vocabulary into a new instance according to a vocabulary of rendering semantics; and *formatting* the instance of the rendering vocabulary in the user agent.

The rendering semantics of much of the XSL vocabulary are device independent, so we can use one set of constructs regardless of the rendering medium. It is the rendering agent's responsibility to interpret these constructs accordingly. In this way, the XSL semantics can be interpreted for print, display, aural or other presentations. There are, indeed, some specialized semantics we can use to influence rendering on particular media, though these are just icing on the cake.

1.1.5 Extensible Stylesheet Language Transformations (XSLT)

We all have needs to transform our structured information when it is not appropriately ordered for a purpose other than how it is created. The XSLT 1.0 Recommendation describes a transformation instruction vocabulary of constructs that can be expressed in an XML model of elements and attributes:

- http://www.w3.org/TR/xslt

1.1.5.1 Transformation by example

We can characterize XSLT from other techniques for transmuting our information by regarding it simply as "Transformation by Example", differentiating many other techniques as "Transformation by Program Logic". This perspective focuses on the distinction that our obligation is not to tell an XSLT processor how to effect the changes we need, rather, we tell an XSLT processor what we want as an end result, and it is the processor's responsibility to do the dirty work.

The XSLT Recommendation gives us a vocabulary for specifying templates that function as "examples of the result". Based on how we instruct the XSLT processor to access the source of the data being transformed, the processor will incrementally build the result by adding the filled-in templates.

We write our stylesheets, or "transformation specifications", primarily with declarative constructs though we can employ procedural techniques if and when needed. We assert the desired behavior of the XSLT processor based on conditions found in our source. We supply examples of how each component of our result is formulated and indicate the conditions of the source that trigger which component is next added to our result. Alternatively we can selectively add components to the result on demand.

Consider once again the customer information in our example purchase order at Example 1-1. An example of the HTML vocabulary supplied to the XSLT processor to produce the markup in Example 1-7 would be:

```
<xsl:template match="customer">
  <p><xsl:text>From: </xsl:text>
    <i><xsl:text>(Customer Reference) </xsl:text>
      <b><xsl:value-of select="@db"/></b></i></p>
</xsl:template>
```

Example 1-9: Example XSLT template rule for the HTML vocabulary

An example of XSL vocabulary supplied to the XSLT processor to produce the markup in Example 1-8 would be:

```
<xsl:template match="customer">
  <fo:block space-before.optimum="20pt" font-size="20pt">
    <xsl:text>From: </xsl:text>
    <fo:inline-sequence font-style="italic">
      <xsl:text>(Customer Reference) </xsl:text>
      <fo:inline-sequence font-weight="bold">
        <xsl:value-of select="@db"/>
      </fo:inline-sequence></fo:inline-sequence></fo:block>
</xsl:template>
```

Example 1-10: Example XSLT template rule for the XSL vocabulary

Where XSLT is similar to other transmutation approaches is that we deal with our information as trees of abstract nodes. We don't deal with the raw syntax of our source data. Unlike these other approaches, however, the primary memory management and information manipulation (node traversal and node creation) is handled by the XSLT processor not by the stylesheet writer. This is a significant difference between XSLT and a transformation programming language or interface like the Document Object Model (DOM), where the programmer is responsible for handling the low-level manipulation of information constructs.

XSLT includes constructs which we use to identify and iterate over structures found in the source information. The information being transformed can be traversed in any order needed and as many times as required to produce the desired result. We can visit source information numerous times if the result of transformation requires that information to be present numerous times.

We users of XSLT don't have the burden of implementing numerous practical algorithms required to present information. The designers of XSLT have specified that such algorithms be implemented within the processor itself, and have enabled us to engage these algorithms declaratively. High-level functions such as sorting and counting are available to us on demand when we need them. Low-level functions such as memory-management, node manipulation and garbage collection are all integral to the XSLT processor.

This declarative nature of the stylesheet markup makes XSLT so very much more accessible to non-programmers than the imperative nature of procedurally-oriented transformation languages. Writing a stylesheet is as simple as using markup to declare the behavior of the XSLT processor, much like HTML is used to declare the behavior of the web browser to paint information on the screen.

The designers have also accommodated the programmer as well as the non-programmer in that there are procedural constructs specified. XSLT is (in theory) "Turing complete", thus any arbitrarily complex algorithm could (theoretically) be implemented using the constructs available. While there will always be a trade-off between extending the processor to implement something internally and writing an elaborate stylesheet to implement something portably, there is sufficient expressive power to implement some algorithmic business rules and semantic processing in the XSLT syntax.

In short, straightforward and common requirements can be satisfied in a straightforward fashion, while unconventional requirements can be satisfied to an extent as well with some programming-styled effort.

Note 4: Theory aside, the necessarily verbose XSLT syntax dictated by its declarative nature and use of XML syntax makes the coding of some complex algorithms a bit awkward. I have implemented some very complex traversals and content generation with successful results, but with code that could be difficult to maintain (my own valiant, if not always satisfactory, documentation practices notwithstanding).

The designers of XSLT recognized the need to maintain large transformation specifications, and the desire to tap prior accomplishments when writing stylesheets so they have included a number of constructs supporting the management, maintenance and exploitation of existing stylesheets. Organizations can build libraries of stylesheet components for sharing among their colleagues. Stylesheet writers can tweak the results of a transformation by writing shell specifications that include or import other stylesheets known to solve problems they are addressing. Stylesheet fragments can be written for particular vocabulary fragments; these fragments can subsequently be used in concert, as part of an organization's strategy for common information description in numerous markup models.

1.1.5.2 *Not* intended for general purpose XML transformations

It is important to remember that XSLT was designed *primarily for transforming XML vocabularies to the XSL formatting vocabulary*. This doesn't preclude us from using XSLT for other transformation requirements, but it does influence the design of the language and it does constrain some of the functionality from being truly general purpose.

For this reason, the designers *do not* claim XSLT is a general purpose transformation language. However, it is still powerful enough for *most* downstream processing transformation needs, and XSLT stylesheets are often called XSLT transformation scripts because they can be used in many areas not at all related to stylesheet rendering. Consider an electronic commerce environment where transformation is not used for presentation purposes. In this case, the XSLT processor may transform a source instance, which is based on a particular vocabulary, and deliver the results to a legacy application that expects a different vocabulary as input. In other words, we can use XSLT in a non-rendering situation when it doesn't matter what syntax is utilized to represent the content; when only the parsed result of the syntax is material.

An example of using such a legacy vocabulary for the XSLT processor would be:

```
<xsl:template match ="customer">
  <buyer><xsl:value-of select="@db"/></buyer>
</xsl:template>
```

Example 1-11: Example XSLT template rule for a legacy vocabulary

The transformation would then produce the following result acceptable to the legacy application:

```
<buyer>cust123</buyer>
```

Example 1-12: Example legacy vocabulary for customer information

The designers of XSLT have focused on the results of delivering parsed XML information to a rendering agent, or to some other application employing an XML processor as the means to access information in an XML instance. The information being delivered represents the parsed result of working with the entire XML instance and, if supplied, the XML document model. The actual markup within the source XML instance is not considered material to the application. All that counts is the result of having processed the XML instance to find the underlying content the actual markup represents.

By focusing on this parsed result for downstream applications, there is little or no regard in an XSLT stylesheet for the actual XML syntax constructs found within the source input documents, or for the actual XML syntax constructs utilized in the resulting output document. This prevents a stylesheet from being aware of such constructs or controlling how such constructs are used. Any transformation requirement that includes "original markup syntax preservation" would not be suited for XSLT transformations.

Note 5: Is not being able to support "original markup syntax preservation" really a problem? That depends how you regard the original markup syntax used in an XML instance. XML allows you to use various markup techniques to meet identical information representation requirements. If you treat this as merely syntactic sugar for human involvement in the markup process, then it will not be important how information is specifically marked up once it is out of the hands of the human involved. If, however, you are working with transformations where such issues are more than just a sugar coating, and it is necessary to utilize particular constructs based on particular requirements of how the result "looks" in syntactic form, then XSLT will not provide the kind of control you will need.

1.1.5.3 Document model and vocabulary independent

While checking source documents for validity can be very useful for diagnostic purposes, all of the hierarchical relationships of content are based on what is found inside of the instance, not what is found in the document model. The behavior of the stylesheet is specified against the presence of markup in an instance as the *implicit* model, not against the allowed markup prescribed by any *explicit* model. Because of this, an XSLT stylesheet is independent of any Document Type Definition (DTD) or other explicit schema that may have been used to constrain the instance at other stages. This is very handy when working with well-formed XML that doesn't have an explicit document model.

If an explicit document model is supplied, certain information such as attribute types and defaulted values enhance the processor's knowledge of the information found in the input documents. Without this information, the processor can still perform stylesheet processing as long as the absence of the information does not influence the desired results.

Without a reliance on the document model for the instance, we can design a single stylesheet that can process instances of different models. When the models are very similar, much of the stylesheet operates the same way each time and the rest of the stylesheet only processes that which it finds in the sources.

It may be obvious but should be stated for completeness that a given source file can be processed with multiple stylesheets for different purposes. This means, though, that it is possible to successfully process a source file with a stylesheet designed for an entirely different vocabulary. The results will probably be totally inappropriate, but there is nothing inherent to an instance that ties it to a single stylesheet or a set of stylesheets. Stylesheet designers might well consider how their stylesheets could validate input; perhaps issuing error messages when unexpected content arrives. However, this is a matter of practice and not a constraint.

1.1.5.4 XML source and stylesheet

The input files to an XSLT processor are one or more stylesheet files and one or more source files. The initial inputs are a single stylesheet file and a single source file. Other stylesheet files are assimilated before the first source file is processed. The XML processor will then access other source files according to the first file's XML content. The XSLT processor may then access other source files at any time under stylesheet control.

All of the inputs must be well-formed (but not necessarily valid) XML documents. This precludes using an HTML file following non-XML lexical conventions, but does not rule out processing an Extensible Hypertext Markup Language (XHTML) file as an input. Many users of existing HTML files that are not XML compliant will need to manipulate or transform them; all that is needed to use XSLT for this is a preprocess to convert existing Standard Generalized Markup Language (SGML) markup conventions into XML markup conventions.

XHTML can be created from HTML using a handy free tool on the W3C site: `http://www.w3.org/People/Raggett/tidy/`. This tool corrects whatever improperly coded HTML it can and flags any that it cannot correct. When the output is configured to follow XML lexical conventions, the resulting file can be used as an input to the XSLT processor.

1.1.5.5 Validation unnecessary (but convenient)

That an XSLT processor need not incorporate a validating XML processor to do its job does not minimize the importance of source validation when developing a stylesheet. Often when working incrementally to develop a stylesheet by simultaneously working on the test source file and stylesheet algorithm, time can be lost by inadvertently introducing well-formed but invalid source content. Because there is no validation in the XSLT processor, all well-formed source will be processed without errors, producing a result based on the data found. The first reaction of the stylesheet writer is often that a problem has been introduced in the stylesheet logic, when in fact the stylesheet works fine for the intended source data. The real problem is that the source data being used isn't as intended.

Note 6:	Personally, I run a separate post-process source file validation after running the source file through a given stylesheet. While I am examining the results of stylesheet processing, the post process determines whether or not the well-formed file validates against the model to which I'm designing the stylesheet. When anomalies are seen I can check the validation for the possible source of a problem before diagnosing the stylesheet itself.

1.1.5.6 Multiple source files possible

The first source file fed to the XSLT processor defines the first abstract tree of nodes the stylesheet uses.

The stylesheet may access arbitrary other source files, or even itself as a source file, to supplement the information found in the primary file. The names of these supplementary resources can be hardwired into the stylesheet, passed to the stylesheet as a parameter, or the stylesheet can find them in the source files.

A separate node tree represents every resource accessed as a source file, each with its own scope of unique node identifiers and global values. When a given resource is identified more than once as a source file, the XSLT processor creates only a single representation for that resource. In this way a stylesheet is guaranteed to work unambiguously with source information.

1.1.5.7 Stylesheet supplements source

A given transformation result does not necessarily obtain all of its information from the source files. It is often (almost always) necessary to supplement the source with boilerplate or other hardwired information. The stylesheet can add any arbitrary information to the result tree as it builds the result tree from information found in the source trees.

A stylesheet can be the synthesis of the primary file and any number of supplemental files that are included or imported by the main file. This provides powerful mechanisms for sharing and exploiting fragments of stylesheets in different scenarios.

1.1.5.8 Extensible language design supplements processing

The "X" in XSLT stands for "Extensible" for a reason: the designers have built-in conforming techniques for accessing non-conforming facilities requested by a stylesheet writer that may or may not be available in the XSLT processor interpreting the stylesheet. A conforming processor may or may not support such extensions and is only obliged to accommodate error and fallback processing in such a way that a stylesheet writer can reconcile the behavior if needed.

An XSLT processor can implement extension instructions, functions, serialization conventions and sorting schemes that provide functionality beyond what is defined in XSLT 1.0, all accessed through standardized facilities.

A stylesheet writer must not rely on any extension facilities if the XSLT processor being used for the stylesheet is not known or is outside of the stylesheet writer's control. If an end-user base utilizes different brands of XSLT processors, and the stylesheet needs to be portable across all processors, only the standardized facilities can be used.

Standardized presence-testing and fallback facilities can be used by the stylesheet writer to accommodate the ability of a processor to act on extension facilities used in the stylesheet.

1.1.5.9 Abstract structure result

In the same way our stylesheets are insulated from the syntax of our source files, our stylesheets are insulated from the syntax of our result.

We do not focus on the syntax of the file to be produced by the XSLT processor; rather, we create a result tree of abstract nodes, which is similar to the tree of abstract nodes of our input information. Our examples of transformation (converted to nodes from our stylesheet) are added to the result hierarchy as nodes, not as syntax. Our objective as XSLT transformation writers is to create a result node tree that may or may not be serialized externally as markup syntax.

The XSLT processor is not obliged to externalize the result tree if the processor is integral to some process interpreting the result tree for other purposes. For example, an XSL rendering agent may embed an XSLT processor for interpreting the inputs to produce the intermediate hierarchy of XSL rendering vocabulary to be reified in a given medium. In such cases, serializing the intermediate tree in syntax is not material to the process of rendering (though having the option to serialize the hierarchy is a useful diagnostic tool).

The stylesheet writer has little or no control over the constructs chosen by the XSLT processor for serializing the result tree. There are some behaviors the stylesheet can request of the processor, though the processor is not obliged to respect the requests. The stylesheet can request a particular output method be used for the serialization and, if supported, the processor guarantees the final result complies with the lexical requirements of that method.

Note 7:	It is possible to coerce the XSLT processor to violate the lexical rules through certain stylesheet controls that I personally avoid using at all costs. For every XML and HTML instance construct (not including the document model syntax constructs) there are proper XSLT methodologies to follow, though not always as compact as coercing the processor.

The abstract nature of the node trees representing the input source and stylesheet instances and the hands-off nature of serializing the abstract result node tree are the primary reasons that source tree original markup syntax preservation cannot be supported.

The design of the language does, however, support the serialization of the result tree in such a way as not to require the XSLT processor to maintain the result tree in the abstract form. For example, the processor can instantly serialize the start of an element as soon as the element content of the result is defined. There is no need to maintain, nor is there any ability in the stylesheet to add to, the start of an element once the stylesheet begins supplying element content.

The XSLT 1.0 Recommendation defines three output methods for lexically reifying the abstract result tree as serialized syntax: XML conventions, HTML conventions, and simple text conventions. An XSLT processor can be extended to support custom serialization methods for specialized needs.

1.1.5.10 Result-tree-oriented objective

This result abstraction impacts how we design our stylesheets. We have to always remember that the result of transformation is created in result parse order, thus allowing the XSLT processor to immediately serialize the result without maintaining the result for later purposes.

The examples of transformation that we include in our stylesheet already represent examples of the nodes that we want added to the result tree, but we must ensure these examples are triggered to be added to the result tree in result parse order, otherwise we will not get the desired result.

We can peruse and traverse our source files in any predictable order we need to produce the result, but we can only produce the result tree once and then only in result tree parse order. It is often difficult to change traditional perspectives of transformation that focus on the source tree, yet we must look at XSLT transformations focused on the result tree.

The predictable orders we traverse the source trees are not restricted to only source tree parse order (also called document order). Information in the source trees can be ignored or selectively processed. The order of the result tree dictates the order in which we must access our source trees.

Note 8:	I personally found this required orientation difficult to internalize, having been focused on the creation of my source information long before addressing issues of transforming the sources to different results. Understanding this orientation is key to quickly producing results using XSLT.

It is not, however, an XSLT processor implementation constraint to serially produce the result tree. This is an important distinction in the language design that supports parallelism. An XSLT processor supporting parallelism can simultaneously produce portions of the result tree provided only that the end result is created as if it were produced serially.

1.1.6 Namespaces

To successfully use and distinguish element types in our instances as being from given vocabularies, the Namespaces in XML Recommendation gives us means to preface our element type names to make them unique. The Recommendation and the following widely-read discussion document describe the precepts for using this technique:

- `http://www.w3.org/TR/REC-xml-names`

1.1.6.1 Vocabulary distinction

It would be unreasonable to mandate that all document models have mutually unique element type names. We design our document models with our own business requirements and our own naming conventions; so do other users. A W3C working group developing vocabularies has its own conventions and requirements; so do other committees. An XML-based application knowing that an instance is using element types from only a single vocabulary can easily distinguish all elements by the name, since each element type is declared in the model by its name.

But what happens when we need to create an XML instance that contains element types from more than one vocabulary? If all the element types are uniquely named then we could guess the vocabulary for a given element by its name. But if the same name is used in more than one vocabulary, we need a technique to avoid ambiguity. Using cryptically compressed or unmanageably elongated element type names to guarantee uniqueness would make XML difficult to use and would only delay the problem to the point that these weakened naming conventions would still eventually result in vocabulary collisions.

Note 9:	Enter the dreaded namespaces: a Recommendation undeserving of its sullied reputation. This is a powerful, yet very simple technique for disambiguating element type names in vocabularies. Perhaps the reputation spread from those unfamiliar with the requirements being satisfied. Perhaps concerns were spread by those who made assumptions about the values used in namespace declarations. As unjustified as it is, evoking namespaces unnecessarily (and unfortunately) strikes fear in many people. It is my goal to help the reader understand that not only are namespaces easy to define and easy to use, but that they are easy to understand and are not nearly as complex as others have believed.

The Namespaces in XML Recommendation describes a technique for exploiting the established uniqueness of Uniform Resource Identifier (URI) values under the purview of the Internet Engineering Task Force (IETF). We users of the Internet accept the authority of the registrar of Internet domain names to allot unique values to organizations, and it is in our best interest to not arrogate or usurp values allotted to others as our own. We can, therefore, assume a published URI value belongs to the owner of the domain used as the basis of the value. The value is not a Uniform Resource Locator (URL), which is a URI that identifies an actual addressed location on the Internet; rather, the URI is being used merely as a unique string value.

To set the stage for how these URI values are used, consider an example of two vocabularies that could easily be used together in an XML instance: the Scalable Vector Graphics (SVG) vocabulary and the Mathematical Markup Language (MathML). In SVG the `<set>` element type is used to scope a value for reference by descendent elements. In MathML the `<set>` element type defines a set in the mathematical sense of a collection.

Remembering that names in XML follow rigid lexical constraints, we pick out of thin air a prefix we use to distinguish each element type from their respective vocabulary. The prefix we choose is *not* mandated by any organization or any authority; in our instances we get to choose any prefix we wish. We should, however, make the prefix meaningful or we will obfuscate our information, so let's choose in this example to distinguish the two element types as `<svg:set>` and `<math:set>`. Note that making the prefix short is a common convention supporting human legibility, and using the colon "`:`" separating the prefix from the rest of the name is prescribed by the Namespaces in XML recommendation.

While we are talking about names, let's not forget that some Recommendations utilize the XML name lexical construct for other purposes, such as naming facilities that may be available to a processor. We get to use this namespace prefix we've chosen on these names to guarantee uniqueness, just as we have done on the names used to indicate element types.

1.1.6.2 URI value association

But having the prefix is not enough because we haven't yet guaranteed global identity or singularity by a short string of name characters; to do so we must associate the prefix with a globally unique URI before we use that prefix. Note that we are unable to use a URI directly as a prefix because the lexical constraints on a URI are looser than those of an XML name; the invalid XML name characters in a URI would cause an XML processor to balk.

We assert the association between a namespace prefix and a namespace URI by using a namespace declaration attribute as in the following examples:

- `xmlns:svg="http://www.w3.org/2000/svg-20000629"`

- `xmlns:math="http://www.w3.org/1998/Math/MathML"`

As noted earlier, the prefix we choose is arbitrary and can be any lexically valid XML name. The prefix is discarded by the namespace-aware processor, and is immaterial to the application using the names; it is only a syntactic shortcut to get at the associated URI. The associated URI supplants the prefix in the internal representation of the name value and the application can distinguish the names by the new composite name that would have been illegal in XML syntax. There is no convention for documenting a namespace qualified name using its associated URI, but one way to perceive the uniqueness is to consider our example as it might be internally represented by an application:

- `<{http://www.w3.org/2000/svg-20000629}set>`
- `<{http://www.w3.org/1998/Math/MathML}set>`

The specification of a URI instead of a URL means that the namespace-aware processor will never look at the URI as a URL to accomplish its work. There never need be any resource available at the URI used in a namespace declaration. The URI is just a string and its value is used only as a string and the fact that there may or may not be any resource at the URL identified by the URI is immaterial to namespace processing. The URI does not identify the location of a schema, or a DTD or any file whatsoever when used by a namespace aware processor.

Note 10: Perhaps some of the confusion regarding namespaces is rooted in the overloading of the namespace URI by some Recommendations. These Recommendations require that the URI represent a URL where a particular resource is located, fetched, and utilized to some purpose. This behavior is outside the scope of namespaces and is mandated solely by the Recommendations that require it.

Practice has, however, indicated an end-user-friendly convention regarding the URI used in namespace declarations. The W3C has placed a documentation file at every URL represented by a namespace URI. Requesting the resource at the URL returns an HTML document discussing the namespace being referenced, perhaps a few pointer documents to specifications or user help information, and any other piece of helpful information deemed suitable for the public consumption. This convention should help clear up many misperceptions about the URI being used to obtain some kind of machine-readable resource or schema, though it will not dispel the misperception that there needs to be some resource of some kind at the URL represented by a namespace URI.

So now a processor can unambiguously distinguish an element's type as being from a particular vocabulary by knowing the URI associated with the vocabulary. Our choice of prefix is arbitrary and of no relevance. The URI we have associated with the prefix used in a namespace-qualified XML name (often called a QName) informs the processor of the identity of the name. Our choice of prefix is used and then discarded by the processor, while the URI persists and is the basis of namespace-aware processing. We have achieved uniqueness and identity in our element type names and other XML names in a succinct legible fashion without violating the lexical naming rules of XML.

1.1.6.3 Namespaces in XSL and XSLT

Namespaces identify different constructs for the processors interpreting XSL formatting specifications and XSLT stylesheets.

An XSL rendering agent responsible for interpreting an XSL formatting specification will recognize those constructs identified with the `http://www.w3.org/1999/XSL/Format` namespace. Note that the year value used in this URI value is not used as a version indictor; rather, the W3C convention for assigning namespace URI values incorporates the year the value was assigned to the working group.

An XSLT processor responsible for interpreting an XSLT stylesheet recognizes instructions and named system properties using the `http://www.w3.org/1999/XSL/Transform` namespace. An XSLT processor will not recognize using an archaic value for working draft specifications of XSLT.

XSLT use namespace-qualified names to identify extensions that implement non-standardized facilities. A number of kinds of extensions can be defined in XSLT including functions, instructions, serialization methods, sort methods and system properties.

The XT XSLT processor written by James Clark is an example of a processor implementing extension facilities. XT uses the `http://www.jclark.com/xt` namespace to identify the extension constructs it implements. Remembering that this is a URI and not a URL, you will not find any kind of resource or file when using this value as a URL.

We also use our own namespaces in an XSLT stylesheet for two other purposes. We need to specify the namespaces of the elements and attributes of our result if the process interpreting the result relies on the vocabulary to be identified. Furthermore, our own non-default namespaces distinguish internal XSLT objects we include in our stylesheets. Each of these will be detailed later where such constructs are described.

1.1.7 Stylesheet association

When we wish to associate with our information one or more preferred or suitable stylesheet resources geared to process that information, the W3C stylesheet association Recommendation describes the syntax and semantics for a construct we can add to our XML documents:

- `http://www.w3.org/TR/xml-stylesheet`

1.1.7.1 Relating documents to their stylesheets

XML information in its authored form is often not organized in an appropriate ordering for consumption. A stylesheet association processing instruction is used at the start of an XML document to indicate to the recipient which stylesheet resources are to be used when reading the contents of that document.

The recipient is not obliged to use the resources referenced and can choose to examine the XML using any stylesheet or transformation process they desire by ignoring the preferences stated within. Some XML applications ignore the stylesheet association instruction entirely, while others choose to steadfastly respect the instruction without giving any control to the recipient. A flexible application will let the recipient choose how they wish to view the content of the document.

The designers of this specification adopted the same semantics of the `<LINK>` construct defined in the HTML 4.0 recommendation:

- `<LINK REL="stylesheet">`
- `<LINK REL="alternate stylesheet">`

1.1.7.2 Ancillary markup

A processing instruction is ancillary to the XML document model constraining the creation and validation of an instance. Therefore, we do not have to model the presence of this construct when we design our document model. Any instance can have any number of stylesheet associations added into the document during or after creation, or even removed, without impacting on the XML content itself.

An application respecting this construct will process the document content with the stylesheet before delivering the content to the application logic. Two cases of this are the use of a stylesheet for rendering to a browser canvas and the use of a transformation script at the front end of an e-commerce application.

The following two examples illustrate stylesheet associations that, respectively, reference an XSL resource and a Cascading Stylesheet (CSS) resource:

```
<?xml-stylesheet href="fancy.xsl" type="text/xsl"?>
```

Example 1-13: Associating an XSL stylesheet

```
<?xml-stylesheet href="normal.css" type="text/css"?>
```

Example 1-14: Associating a CSS stylesheet

The following example naming the association for later reference and indicating that it is not the primary stylesheet resource is less typical, but is allowed for in the specification:

```
<?xml-stylesheet alternate="yes" title="small"
                 href="small.xsl" type="text/xsl"?>
```

Example 1-15: Alternative stylesheet association

A URL that does not include a reference to another resource, but rather is defined exclusively by a local named reference, specifies a stylesheet resource that is located inside the XML document being processed, as in the following example:

```
<?xml-stylesheet href="#style1" type="text/xsl"?>
```

Example 1-16: Associating an internal stylesheet

The Recommendation designers expect additional schemes for linking stylesheets and other processing scripts to XML documents to be defined in future specifications.

Note 11:	Embedding stylesheet association information in an XML document and using the XML processing instruction to do so are both considered stopgap measures by the W3C. This Recommendation cautions readers that no precedents are set by employing these makeshift techniques and that urgency dictated their choice. Indeed, there is some question as to the appropriateness of tying processing to data so tightly, and we will see what considered approaches become available to us in the future.

This is a prose version of an excerpt from the book "Practical Transformation Using XSLT and XPath" (Eighth Edition ISBN 1-894049-05-5 at the time of this writing) published by Crane Softwrights Ltd., written by G. Ken Holman; this excerpt was edited by Stan Swaren, and reviewed by Dave Pawson.

1. The Context of XSL Transformations and the XML Path Language (cont'd)

1.2 Transformation data flows

Here we look at the interactions between some of the Recommendations we focus on by examining how our information flows through processes engaging or supporting the technologies.

1.2.1 Transformation from XML to XML

As we will see when looking at the data model, the normative behavior of XSLT is to transform an XML source into an abstract hierarchical result.

We can request that result to be serialized into an XML file, thus we achieve XML results from XML sources:

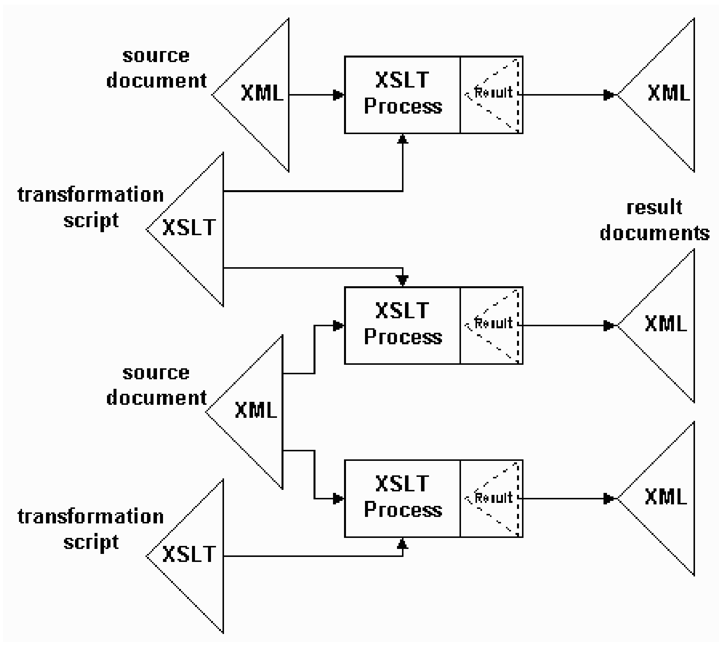


Figure 1-3: Transformation from XML to XML

An XSLT stylesheet can be applied to more than one XML document, each stylesheet producing a possibly (usually) different result. Nothing in XSLT inherently ties the stylesheet to a single instance, though the stylesheet writer can employ techniques to abort processing based on processing undesirable input.

An XML document can have more than one XSLT stylesheet applied, each stylesheet producing a possible (usually) different result. Even when stylesheet association indicates an author's preference for a stylesheet to use for processing, tools should provide the facility to override the preference with the reader's preference for a stylesheet. Nothing in XML prevents more than a single stylesheet to be applied.

Note 12:	In all cases in this chapter the depictions show the normative result of the XSLT processor's as the dotted triangle attached to the process rectangle. This serves to remind the reader that the serialization of the result into an XML file is a separate task, one that is the responsibility of the XSLT processor and not the stylesheet writer. In all diagrams, the left-pointing triangle represents a hierarchically-marked up document such as an XML or HTML document. This convention stems from considering the apex of the hierarchy at the left, with the sub-elements nesting within each other towards the lowest leaves of the hierarchy at the right of the triangle. Processes are depicted in rectangles, while arbitrary data files of some binary or text form are depicted in parallelograms. Other symbols representing screen display, print and auditory output are drawn with (hopefully) obvious shapes.

1.2.2 Transformation from XML to XSL formatting semantics

When the result tree is specified to utilize the XSL formatting vocabulary, the normative behavior of an XSL processor incorporating an XSLT processor is to interpret the result tree. This interpretation reifies the semantics expressed in the constructs of the result tree to some medium, be it pixels on a screen, dots on paper, sound through a synthesis device, or another medium that makes sense for presentation.

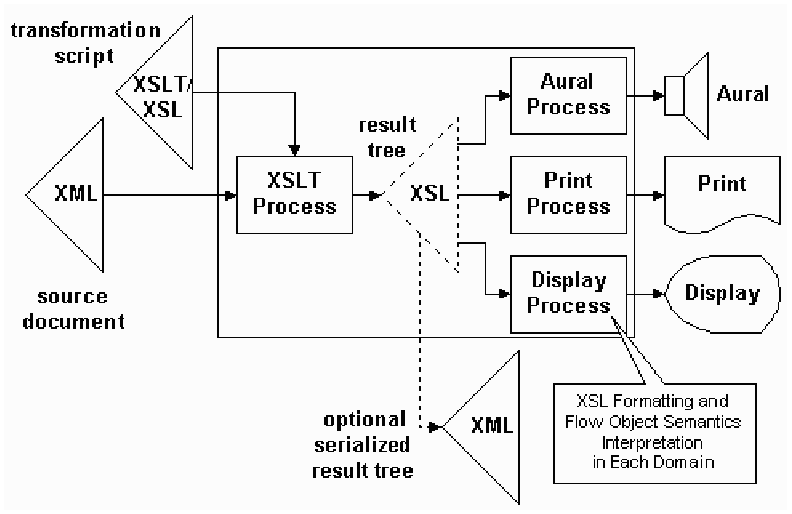


Figure 1-4: Transformation from XML to XSL Formatting Semantics

Without employing extension techniques or supplemental documentation, the stylesheets used in this scenario contain *only* the transformation vocabulary and the resulting formatting vocabulary. There are no other element types from other vocabularies in the result, including from the source vocabulary. For example, rendering processors would not inherently know what to do with an element of type `custnbr` representing a customer number; it is the stylesheet writer's responsibility to transform the information into information recognized by the rendering agent.

There is no obligation for the rendering processor to serialize the result tree created during transformation. The feature of serializing the result tree to XML syntax is, however, quite useful as a diagnostic tool, revealing to us what we really asked to be rendered instead of what we thought we were asking to be rendered when we saw incorrect results. There may also be performance considerations of taking the reified result tree in XML syntax and rendering it in other media without incurring the overhead of performing the transformation repeatedly.

1.2.3 Transformation from XML to non-XML

An XSLT processor may choose to recognize the stylesheet writer's desire to serialize a non-XML representation of the result tree:

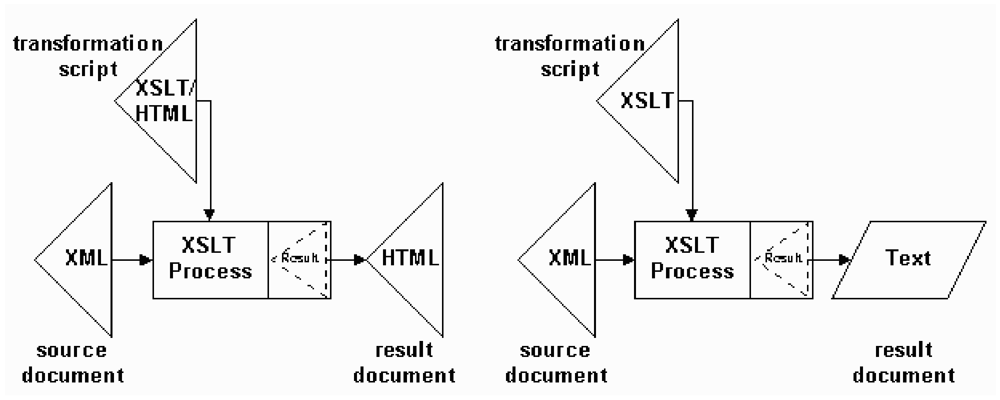


Figure 1-5: Transformation from XML to Aware Non-XML

The XSLT Recommendation documents two non-XML tree serialization methods that can be requested by the stylesheet writer. When the processor offers serialization, it is *only* obliged to reify the result using XML lexical and syntax rules, and *may* support producing output following either HTML lexical and syntax rules or simple text.

1.2.3.1 HTML lexical and syntactic conventions

Internet web browsers are specific examples of the generic HTML user agent. User agents are typically designed for instances of HTML following the precursor to XML: the Standard Generalized Markup Language (SGML) lexical conventions. Certain aspects of the HTML document model also dictate syntactic shortcuts available when working with SGML.

While some more recently developed user agents will accept XML lexical conventions, thus accepting Extensible Hypertext Markup Language (XHTML) output from an XSLT processor, older user agents will not. Some of these user agents will not accept XML lexical conventions for empty elements, while some require SGML syntax minimization techniques to compress certain attribute specifications.

Additionally, user agents recognize a number of general entity references as built-in characters supporting accented letters, the non-breaking space, and other characters from public entity sets defined or used by the designers of HTML. An XSLT processor recognizes the use of these characters in the result tree and serializes them using the assumed built-in general entities.

1.2.3.2 Text lexical conventions

An XSLT processor can be asked to serialize only the `#PCDATA` content of the entire result tree, resulting in a file of simple text without employing any markup techniques. All text is represented by the characters' individual values, even those characters sensitive to XML interpretation.

Note 13:	I use the text method often for synthesizing MSDOS batch files. By walking through my XML source I generate commands to act on resources identified therein, thus producing an executable batch file tailored to the information.

1.2.3.3 Arbitrary binary and custom lexical conventions

Many of our legacy systems or existing applications expect information to follow custom lexical conventions according to arbitrary rules. Often, this format is raw binary not following textual lexical patterns. We are usually obliged to write custom programs and transformation applications to convert our XML information to these non-standardized formats due to their binary or non-structured natures.

XSLT can play a role even here where the target format is neither structured, nor text, nor in any format anticipated by the designers of the Recommendation. We do have a responsibility to fill in a critical piece of the formula described below, but we can leverage this single effort in the equation to allow us and our colleagues to continue to use W3C Recommendations with our XML data.

Not using XSLT to produce custom output

Consider first the scenario without using XSLT where we must write individual XML-aware applications to accommodate our information vocabularies. For each of our vocabularies we need *separate* programs to convert to the common custom format required by the application. This incurs programming resources to accommodate any and every change to our vocabularies in order to meet the new inputs to satisfy the same custom output.

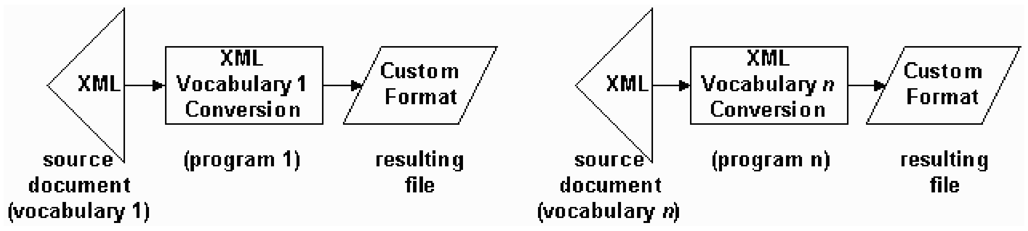

XML
XML Vocabulary 1 Conversion
Custom Format
source document (vocabulary 1)
(program 1)
resulting file
XML
XML Vocabulary n Conversion
Custom Format
source document (vocabulary n)
(program n)
resulting file

Figure 1-6: Accommodating multiple inputs with different XML vocabularies

Using XSLT to produce custom output

If, however, we focus on the custom output instead of focusing on our vocabulary inputs, we can leverage a single investment in programming across all of our vocabularies. Moreover, by being independent of the vocabulary used in the source, we can accommodate any of our or others' vocabularies we may have to deal with in the future.

The approach involves us creating our own custom markup language based on a critical analysis of the target custom format to distill the semantics of how information is represented in the resulting file. These semantics can be expressed using an XML vocabulary whose elements and attributes engage the features and functions of the resulting format. We must not be thinking of our source XML vocabularies, rather, our focus is entirely on the semantics of what exactly makes up our target custom format. Let's refer to this custom format's XML vocabulary we divine from our analysis as the Custom Vocabulary Markup Language (CVML).

Using our programming resources we can then write a single transformation application responsible for interpreting XML instances of CVML to produce a file following the custom format. This transformation application could be written using the Document Object Model (DOM) as a basis for tree-oriented access to the information. Alternatively, a SAX-based application can interpret the instances to produce the outputs if the nature of CVML lends itself to that orientation. The key is that regardless of how instances of CVML are created, the interpretation of CVML markup to produce an output file never changes. Our *one* CVML Instance Interpreter application can produce *any* custom format output file expressible in the CVML semantics.

Getting back to our own or others' XML vocabularies, we have now reduced the problem to XML instance transformation. Our objective is simplified to produce XML instances of CVML from instances of our many input XML vocabularies. This is a classical XSLT situation and we need only write XSLT stylesheets combining the XSLT instructions with CVML as the result vocabulary. Our investment in XSLT for our colleagues is leveraged by the CVML Instance Interpreter so that they can now take their XML and use stylesheets to produce the binary or custom lexical format.

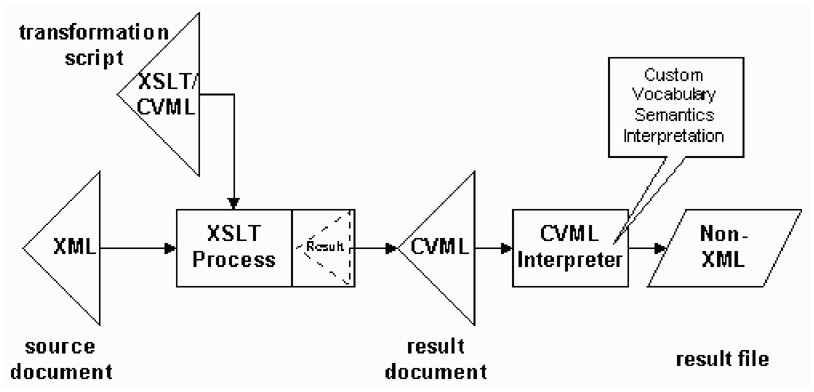


Figure 1-7: Transformation from XML to an arbitrary format

This approach separates the awareness of the lexical and syntactic requirements of the custom output format from the numerous stylesheets we write for all of our possible input XML vocabularies. Our colleagues use XSLT just as they would with HTML or XSL as a result vocabulary. They leverage the single investment in producing the custom format by using the CVML Interpreter to serialize the results of their transformations to produce the files designed for other applications. This, in turn, leverages the investment in learning and using XSLT in the organization.

Taking this two steps further

First, the "X" in XSLT represents the word "extensible" and result tree serialization is one of the areas where we can extend an XSLT processor's functionality. This allows us to implement non-standard vendor-specific or application-specific output serialization methods and engage these facilities in a standard manner. As with all extension mechanisms in XSLT, the trigger is the use of an XML namespace recognized by the XSLT processor implementing the extension:

```
01  xmlns:prefix="processor-recognized-URI"
02  <xsl:output method="prefix:serialization-method-name"/>
```

Example 1-17: Using namespaces to specify an extension serialization method

Comment:
- the namespace declaration attribute on line 1 must be somewhere in the element or the ancestry of the instruction on line 2

Using the same semantics described for the outboard CVML Interpreter program depicted in Figure 1-7, this translation facility can be incorporated into the XSLT processor itself as an inboard extension. The code itself may be directly portable based on the nature of how the outboard program is written. Such an extended processor would directly emit the custom format without reifying the intermediate structure (though this would be convenient for diagnostic purposes):

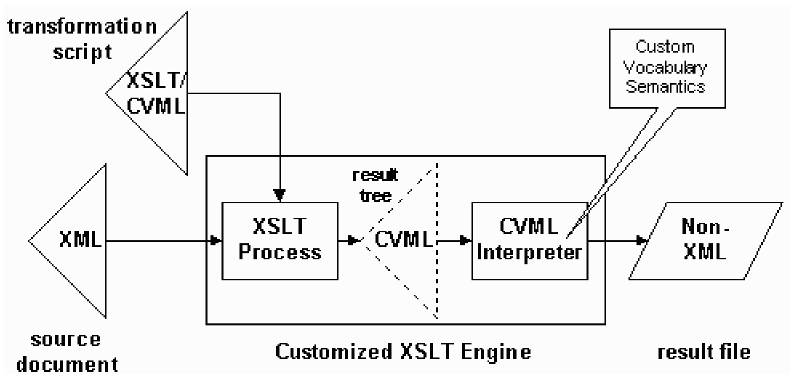


Figure 1-8: Built-in Transformation from XML to Arbitrary Non-XML

The XT XSLT processor implements an extension serialization method named NXML for "non-XML":

```
01  xmlns:prefix="http://www.jclark.com/xt"
02  <xsl:output method="prefix:nxml"/>
```

Example 1-18: Using the XT namespace to specify the NXML extension serialization method

Comment:
- the namespace declaration attribute on line 1 must be somewhere in the element or the ancestry of the instruction on line 2

Second, this extensibility opens up the opportunity to use an XSLT processor as a front-end to any application that can be modified to access the result tree. The intermediate result tree of CVML is not serialized externally; rather, it is fed directly to the application and the application interprets the internal representation of the content that would have been serialized to a custom format. Time is saved by not serializing the result tree and having the application parse the reified file back into a memory representation; performance is enhanced by the application directly accessing the result of transformation.

When generalized, a vendor's non-XML-based application can use this approach to accommodate arbitrary customers' XML vocabularies merely by writing W3C conforming XSLT stylesheets as the "interpretation specification". Some XSLT processors can build a DOM representation of result tree or deliver the result tree as Simple API for XML (SAX) events, thus giving an application developer standardized interfaces to the transformed information expressed using the application's custom semantics vocabulary. The developer's programming is then complete and the vendor accommodates each customer vocabulary with an appropriate stylesheet for translation to the application semantics.

1.2.4 Three-tiered architectures

A three-tiered architecture can meet technical and business objectives by delivering structured information to web browsers by using XSLT on the host, or on the user agent, or even on both.

Considering technical issues first, the server can distribute the processing load to XML/XSLT-aware user agents by delivering a combination of the stylesheet and the source information to be transformed on the recipient's platform. Alternatively, the server can perform the transformations centrally to accommodate those user agents supporting only HTML or HTML/CSS vocabularies:

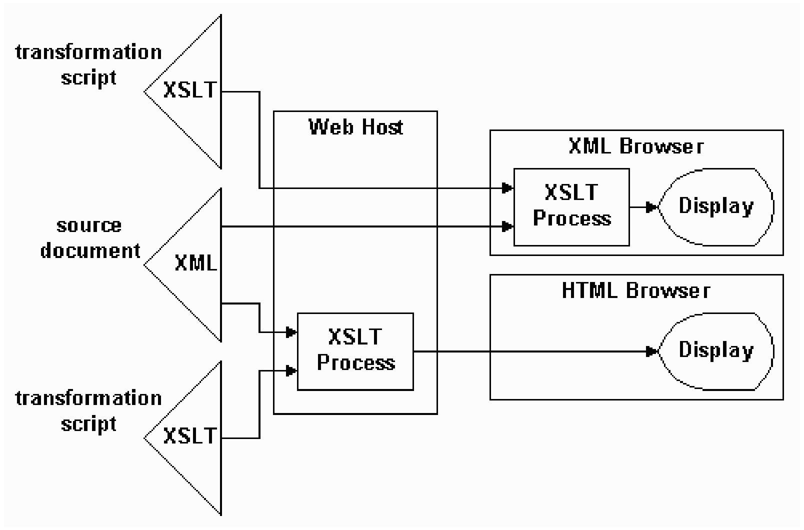


Figure 1-9: Server-side Transformation Architecture

There may be good business reasons to selectively deliver richly-marked-up XML to the user agent or to arbitrarily transform XML to HTML on the server regardless of the user agent capabilities. Even if it is technically possible to send semantically-rich information in XML, protecting your intellectual property by hiding the richness behind the security of a "semantic firewall" must be considered. Perhaps there are revenue opportunities by only delivering a richly marked-up rendition of your information to your customers. Perhaps you could even scale the richness to differing levels of utility for customers who value your information with different granularity or specificity, while preserving the most detailed normative version of the data away of view.

Lastly, there are no restrictions to using two XSLT processes: one on the server to translate our organization's rich markup into an arbitrary delivery-oriented markup. This delivery markup, in turn, is translated using XSLT on the user agent for consumption by the operator. This approach can reduce bandwidth utilization and increase distributed processing without sacrificing privacy.

Note 14:	There is no consensus in our XML community that semantic firewalls are a "good thing". Peers of mine preach that the World Wide Web must always be a semantic web with rich markup processed in a distributed fashion among user agents being de rigueur. Personally, I do not subscribe to this point of view. We have the flexibility to weigh the technical and business perspectives of our customers' needs for our information, our own infrastructure and processing capabilities, and our own commercial and privacy concerns. We can choose to "dumb down" our information for consumption, and the installed base of user agents supporting presentation-oriented semantic-less HTML can be the perfect delivery vehicle to protect these concerns of ours.

This is a prose version of an excerpt from the book "Practical Transformation Using XSLT and XPath" (Eighth Edition ISBN 1-894049-05-5 at the time of this writing) published by Crane Softwrights Ltd., written by G. Ken Holman; this excerpt was edited by Stan Swaren, and reviewed by Dave Pawson.

2. Getting started with XSLT and XPath

Examining working stylesheets can help us understand how we use XSLT and XPath to perform transformations. This article first dissects some example stylesheets before introducing basic terminology and design principles.

2.1 Stylesheet examples

Let's first look at some example stylesheets using two implementations of XSLT 1.0 and XPath 1.0: the XT processor from James Clark, and the third web release of Internet Explorer 5's MSXML Technology Preview.

These two processors were chosen merely as examples of, respectively, standalone and browser-based XSLT/XPath implementations, without prejudice to other conforming implementations. The code samples only use syntax conforming to XSLT 1.0 and XPath 1.0 recommendations and will work with any conformant XSLT processor.

Note: The current (4/14/2000) Internet Explorer 5 *production* release supports only an archaic experimental dialect of XSLT based on an early working draft of the recommendation. The examples in this book *will not* run on the production release of IE5. The production implementation of the old dialect is described in `http://msdn.microsoft.com/xml/XSLGuide/conformance.asp`.

2.1.1 Some simple examples

Consider the following XML file `hello.xml` obtained from the XML 1.0 Recommendation and modified to declare an associated stylesheet:

```
01  <?xml version="1.0"?>
02  <?xml-stylesheet type="text/xsl" href="hello.xsl"?>
03  <greeting>Hello world.</greeting>
```

Example 2-1: The first sample instance in XML 1.0 (modified)

We will use this simple file as the source of information for our transformation. Note that the stylesheet association processing instruction in line 2 refers to a stylesheet with the name "`hello.xsl`" of type XSL. Recall that an XSLT processor is *not* obliged to respect the stylesheet association preference, so let us first use a standalone XSLT processor with the following stylesheet `hellohtm.xsl`:

```
01  <?xml version="1.0"?><!--hellohtm.xsl-->
02  <!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
03  <html xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
04        xsl:version="1.0">
05   <head><title>Greeting</title></head>
06   <body><p>Words of greeting:<br/>
07     <b><i><u><xsl:value-of select="greeting"/></u></i></b>
08     </p></body>
09  </html>
```

Example 2-2: An implicitly-declared simple stylesheet

This file looks like a simple XHTML file: an XML file using the HTML vocabulary. Indeed, it is just that, but we are allowed to inject into the instance XSLT instructions using the prefix for the XSLT vocabulary declared on line 3. We can use any XML file as an XSLT stylesheet provided it declares the XSLT vocabulary within and indicates the version of XSLT being used. Any prefix can be used for XSLT instructions, though convention often sees `xsl:` as the prefix value.

Line 7 contains the only XSLT instruction in the instance. The `xsl:value-of` instruction uses an XPath expression in the `select=` attribute to calculate a string value from our source information. XPath views the source hierarchy using parent/child relationships. The XSLT processor's initial focus is the root of the document, which is considered the parent of the document element. Our XPath expression value "`greeting`" selects the child named "`greeting`" from the current focus, thus returning the value of the document element named "`greeting`" from the instance.

Using an MS-DOS command-line invocation to execute the standalone processor, we see the following result:

```
X:\samp>xt hello.xml hellohtm.xsl hellohtm.htm
X:\samp>type hellohtm.htm
<html>
<head>
<title>Greeting</title>
</head>
<body>
<p>Words of greeting:<br>
<b><i><u>Hello world.</u></i></b>
</p>
</body>
</html>
```

```
13
14  X:\samp>
```

Example 2-3: Explicit invocation of Example 2-2

Note how the end result contains a mixture of the stylesheet markup and the source instance content, without any use of the XSLT vocabulary. The processor has recognized the use of HTML by the name of the document element and has engaged SGML lexical conventions.

The SGML lexical conventions are evidenced on line 8 where the `<br>` empty element has been serialized without the XML lexical convention for the closing delimiter. This corresponds to line 6 of our stylesheet in Example 2-2 where this element is marked up as `<br/>` according to XML rules. Our inputs are always XML but the XSLT processor may recognize the output as being HTML and serialize the result following SGML rules.

Consider next the following explicitly-declared XSLT file `hello.xsl` to produce XML output using the HTML vocabulary, thus the output is serialized as XHTML:

```
01  <?xml version="1.0"?><!--hello.xsl-->
02  <!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
03
04  <xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
05                 version="1.0">
06
07  <xsl:output method="xml" omit-xml-declaration="yes"/>
08
09  <xsl:template match="/">
10      <b><i><u><xsl:value-of select="greeting"/></u></i></b>
11  </xsl:template>
12
13  </xsl:transform>
```

Example 2-4: An explicitly-declared simple stylesheet

This file explicitly declares the document element of an XSLT stylesheet with the requisite XSLT namespace and version declarations. Line 7 declares the output to follow XML lexical conventions and that the XML declaration is to be omitted from the serialized result. Lines 9 through 11 declare the content of the result that is added when the source information position matches the XPath expression in the `match=` attribute on line 9. The value of "`/`" matches the root of the document, hence, this refers to the XSLT processor's initial focus.

The result we specify on line 10 wraps our source information in the HTML elements without the boilerplate used in the previous example. Line 13 ends the formal specification of the stylesheet content.

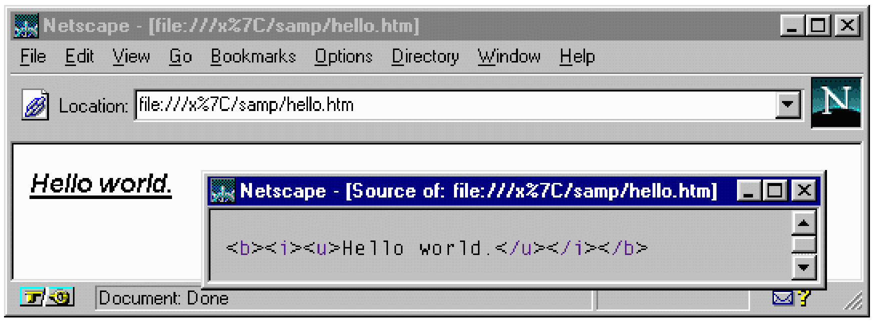
Using an MS-DOS command-line invocation to execute the XT processor we see the following result:

```
X:\samp>xt hello.xml hello.xsl hello.htm

X:\samp>type hello.htm
<b><i><u>Hello world.</u></i></b>
X:\samp>
```

Example 2-5: Explicit invocation of Example 2-4

Using a non-XML-aware browser to view the resulting HTML in Example 2-5 we see the following on the canvas (the child window is opened using the View/Source menu item):

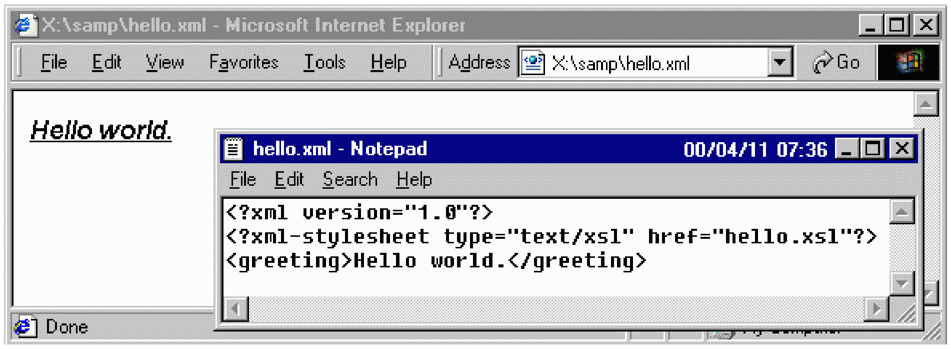

Netscape - [file:///x%7C/samp/hello.htm]
File Edit View Go Bookmarks Options Directory Window Help
Location: file:///x%7C/samp/hello.htm
Netscape - [Source of: file:///x%7C/samp/hello.htm]
Document: Done

Hello world.

```
<b><i><u>Hello world.</u></i></b>
```

Figure 2-1: An non-XML-aware browser viewing the source of a document

Using an XML-aware browser recognizing the W3C stylesheet association processing instruction in Example 2-1, the canvas is painted with the HTML resulting from application of the stylesheet (the child window is opened using the View/Source menu item):


X:\samp\hello.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address X:\samp\hello.xml
Go
Hello world.
hello.xml - Notepad 00/04/11 07:36
File Edit Search Help
Done

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="hello.xsl"?>
<greeting>Hello world.</greeting>
```

Figure 2-2: An XML-aware browser viewing the source of a document

The canvas content matches what the non-XML browser rendered in Figure 2-1. Note that View/Source displays the raw XML source and not the transformed XHTML result of applying the stylesheet.

Note: I found it very awkward when first using browser-based stylesheets to diagnose problems in my stylesheets. Without access to the intermediate results of transformation, it is often impossible to ascertain the nature of the faulty HTML generation. One of the free resources found on the Crane Softwrights Ltd. web site is a script for standalone command-line invocation of the MSXML XSLT processor. This script is useful for diagnosing problems by revealing the result of transformation. This script has also been used extensively by some to create static HTML snapshots of their XML for delivery to non-XML-aware browsers.

This is a prose version of an excerpt from the book Practical Transformation Using XSLT and XPath (Eighth Edition ISBN 1-894049-05-5 at the time of this writing) published by Crane Softwrights Ltd., written by G. Ken Holman; this excerpt was edited by Stan Swaren, and reviewed by Dave Pawson.

2. Getting started with XSLT and XPath (cont'd)

2.1.2 Some more complex examples

The following more complex examples are meant merely as illustrations of some of the powerful facilities and techniques available in XSLT. These samples expose concepts such as variables, functions, and process control constructs a stylesheet writer uses to effect the desired result, but does not attempt any tutelage in their use.

Note: This subsection can be skipped entirely, or, for quick exposure to some of the facilities available in XSLT and XPath, only briefly reviewed. In the associated narratives, I've avoided the precise terminology that hasn't yet been introduced and I overview the stylesheet contents and processor behaviors in only broad terms. Subsequent subsections of this chapter review some of the basic terminology and design approaches.

I hope not to frighten the reader with the complexity of these examples, but it is important to realize that there are more complex operations than can be illustrated using our earlier three-line source file example. The complexity of your transformations will dictate the complexity of the stylesheet facilities being engaged. Simple transformations can be performed quite simply using XSLT, but not all of us have to meet only simple requirements.

The following XML source information in `prod.xml` is used to produce two very dissimilar renderings:

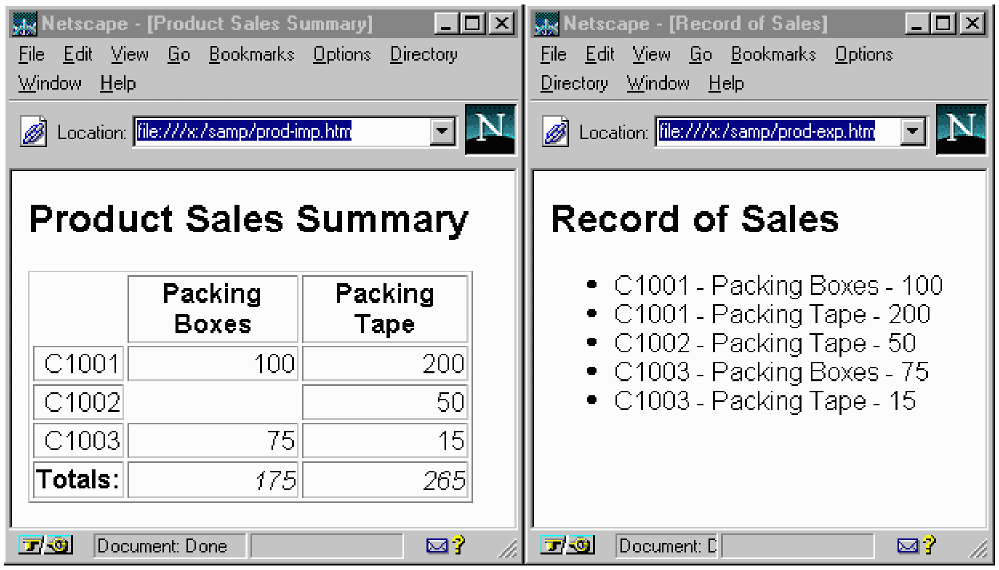
```
01  <?xml version="1.0"?><!--prod.xml-->
02  <!DOCTYPE sales [
03  <!ELEMENT sales ( products, record )> <!--sales information-->
04  <!ELEMENT products ( product+ )>        <!--product record-->
05  <!ELEMENT product ( #PCDATA )>      <!--product information-->
06  <!ATTLIST product id ID #REQUIRED>
07  <!ELEMENT record ( cust+ )>                  <!--sales record-->
08  <!ELEMENT cust ( prodsale+ )>     <!--customer sales record-->
09  <!ATTLIST cust num CDATA #REQUIRED>     <!--customer number-->
10  <!ELEMENT prodsale ( #PCDATA )>     <!--product sale record-->
11  <!ATTLIST prodsale idref IDREF #REQUIRED>
12  ]>
13  <sales>
14    <products><product id="p1">Packing Boxes</product>
15              <product id="p2">Packing Tape</product></products>
16    <record><cust num="C1001">
17              <prodsale idref="p1">100</prodsale>
18              <prodsale idref="p2">200</prodsale></cust>
19            <cust num="C1002">
20              <prodsale idref="p2">50</prodsale></cust>
21            <cust num="C1003">
22              <prodsale idref="p1">75</prodsale>
```

```
23            <prodsale idref="p2">15</prodsale></cust></record>
24  </sales>
```

Example 2-6: Sample product sales source information

Lines 2 through 11 describe the document model for the sales information. Lines 14 and 15 summarize product description information and have unique identifiers according to the `ID`/`IDREF` rules. Lines 16 through 23 summarize customer purchases (product sales), each entry referring to the product having been sold by use of the `idref=` attribute. Not all customers have been sold all products.

Consider the following two renderings of the same data using two orientations, each produced with different stylesheets:


Netscape - [Product Sales Summary]
File Edit View Go Bookmarks Options Directory Window Help
Location: file:///x:/samp/prod-imp.htm
Document: Done
Netscape - [Record of Sales]
File Edit View Go Bookmarks Options Directory Window Help
Location: file:///x:/samp/prod-exp.htm
Document: D

Product Sales Summary

	Packing Boxes	Packing Tape
C1001	100	200
C1002		50
C1003	75	15
Totals:	*175*	*265*

Record of Sales

- C1001 - Packing Boxes - 100
- C1001 - Packing Tape - 200
- C1002 - Packing Tape - 50
- C1003 - Packing Boxes - 75
- C1003 - Packing Tape - 15

Figure 2-3: Different HTML results from the same XML source.

Note how the same information is projected into a table orientation on the left canvas and a list orientation on the right canvas. The one authored order is delivered in two different presentation orders. Both results include titles from boilerplate text not found in the source. The table information on the left includes calculations of the sums of quantities in the columns, generated by the stylesheet and not present explicitly in the source.

The implicit stylesheet `prod-imp.xsl` is an XHTML file utilizing the XSLT vocabulary for instructions to fill in the one result template by pulling data from the source:

```
01  <?xml version="1.0"?><!--prod-imp.xsl-->
02  <!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
03  <html xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
04        xsl:version="1.0">
05    <head><title>Product Sales Summary</title></head>
06    <body><h2>Product Sales Summary</h2>
07      <table summary="Product Sales Summary" border="1">
08                                               <!--list products-->
09        <th align="center">
10            <xsl:for-each select="//product">
11              <td><b><xsl:value-of select="."/></b></td>
12            </xsl:for-each></th>
13                                              <!--list customers-->
14        <xsl:for-each select="/sales/record/cust">
15          <xsl:variable name="customer" select="."/>
16          <tr align="right"><td><xsl:value-of select="@num"/></td>
17            <xsl:for-each select="//product">   <!--each product-->
18              <td><xsl:value-of select="$customer/prodsale
19                                       [@idref=current()/@id]"/>
20              </td></xsl:for-each>
21          </tr></xsl:for-each>
22                                                   <!--summarize-->
23        <tr align="right"><td><b>Totals:</b></td>
24            <xsl:for-each select="//product">
25              <xsl:variable name="pid" select="@id"/>
26              <td><i><xsl:value-of
27                        select="sum(//prodsale[@idref=$pid])"/></i>
28              </td></xsl:for-each></tr>
29      </table>
30    </body></html>
```

Example 2-7: Tabular presentation of the sample product sales source information

Recall that a stylesheet is oriented according to the desired result, producing the result in result parse order. The entire document is an HTML file whose document element begins on line 3 and ends on line 30. The XSLT namespace and version declarations are included in the document element. The naming of the document element as "html" triggers the default use of HTML result tree serialization conventions. Lines 5 and 6 are fixed boilerplate information for the mandatory `<title>` element.

Lines 7 through 29 build the result table from the content. A single header row `<th>` is generated in lines 9 through 12, with the columns of that row generated by traversing all of the `<product>` elements of the source. The focus moves on line 11 to each `<product>` source element in turn and the markup associated with the traversal builds each `<td>` result element. The content of each column is specified as "`.`", which for an element evaluates to the string value of that element.

Having completed the table header, the table body rows are then built, one at a time traversing each `<cust>` child of a `<record>` child of the `<sales>` child of the root of the document, according to the XPath expression "`/sales/record/cust`". The current focus moves to the `<cust>` element for the processing on lines 15 through 21. A local scope variable is bound on line 15 with the tree location of the current focus (note how this instruction uses the same XPath expression as on line 11 but with a different result). A table row is started on line 16 with the leftmost column calculated from the `num=` attribute of the `<cust>` element being processed.

The stylesheet then builds in lines 17 through 20 a column for each of the same columns created for the table header on line 10. The focus moves to each product in turn for the processing of lines 18 through 20. Each column's value is then calculated with the expression "`$customer/prodsale[@idref=current()/@id]`", which could be expressed as follows "from the customer location bound to the variable `$customer`, from all of the `<prodsale>` children of that customer, find that child whose `idref=` attribute is the value of the `id=` attribute of the focus element." When there is no such child, the column value is empty and processing continues. As many columns are produced for a body row as for the header row and our output becomes perfectly aligned.

Finally, lines 23 through 28 build the bottom row of the table with the totals calculated for each product. After the boilerplate leftmost column, line 24 uses the same "`//product`" expression as on lines 10 and 17 to generate the same number of table columns. The focus changes to each product for lines 25 through 28. A local scope variable is bound with the focus position in the tree. Each column is then calculated using a built-in function as the sum of all `<prodsale>` elements that reference the column being totaled. The XPath designers, having provided the `sum()` function in the language, keep the stylesheet writer from having to implement complex counting and summing code; rather, the writer merely declares the need for the summed value to be added to the result on demand by using the appropriate XPath expression.

The file `prod-exp.xsl` is an explicit XSLT stylesheet with a number of result templates for handling source information:

```
01  <?xml version="1.0"?><!--prod-exp.xsl-->
02  <!--XSLT 1.0 - http://www.CraneSoftwrights.com/training -->
03  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
04                  version="1.0">
05
06  <xsl:template match="/">                         <!--root rule-->
07    <html><head><title>Record of Sales</title></head>
08      <body><h2>Record of Sales</h2>
09        <xsl:apply-templates select="/sales/record"/>
10      </body></html></xsl:template>
11
12  <xsl:template match="record">   <!--processing for each record-->
13    <ul><xsl:apply-templates/></ul></xsl:template>
14
15  <xsl:template match="prodsale">   <!--processing for each sale-->
16    <li><xsl:value-of select="../@num"/>   <!--use parent's attr-->
17        <xsl:text> - </xsl:text>
18        <xsl:value-of select="id(@idref)"/>      <!--go indirect-->
19        <xsl:text> - </xsl:text>
20        <xsl:value-of select="."/></li></xsl:template>
21
22  </xsl:stylesheet>
```

Example 2-8: List-oriented presentation of the sample product sales source information

The document element on line 3 includes the requisite declarations of the language namespace and the version being used in the stylesheet. The children of the document element are the template rules describing the source tree event handlers for the transformation. Each event handler associates a template with an event trigger described by an XPath expression.

Lines 6 through 10 describe the template rule for processing the root of the document, as indicated by the "`/`" trigger in the `match=` attribute on line 6. The result document element and boilerplate is added to the result tree on lines 7 and 8. Line 9 instructs the XSLT processor in `<xsl:apply-templates>` to visit all `<record>` element children of the `<sales>` document element, as specified in the `select=` attribute. For each location visited, the processor pushes that location through the stylesheet, thus triggering the template of result markup it can match for each location.

Lines 12 and 13 describe the result markup when matching a `<record>` element. The focus moves to the `<record>` element being visited. The template rule on line 13 adds the markup for the HTML unordered list `<ul>` element to the result tree. The content of the list is created by instructing the processor to visit all children of the focus location (implicitly by not specifying any `select=` attribute) and apply the templates of result markup it triggers for each child. The only children of `<record>` are `<cust>` elements.

The stylesheet does not provide any template rule for the `<cust>` element, so built-in template rules automatically process the children of each location being visited in turn. Implicitly, then, our source information is being traversed in the depth-first order, visiting the locations in parse order and pushing each location through any template rules that are then found in the stylesheet. The children of the `<cust>` elements are `<prodsale>` elements.

The stylesheet does provide a template rule in lines 15 through 20 to handle a `<prodsale>` element when it is pushed, so the XSLT processor adds the markup triggered by that rule to the result. The focus changes when the template rule handles it, thus, lines 16, 18, and 20 each pull information relative to the `<prodsale>` element, respectively: the parent's `num=` attribute (the `<cust>` element's attribute); the string value of the target element being pointed to by the `<prodsale>` element's `idref=` attribute (indirectly obtaining the `<product>` element's value); and the value of the `<prodsale>` element itself.

This is a prose version of an excerpt from the book "Practical Transformation Using XSLT and XPath" (Eighth Edition ISBN 1-894049-05-5 at the time of this writing) published by Crane Softwrights Ltd., written by G. Ken Holman; this excerpt was edited by Stan Swaren, and reviewed by Dave Pawson.

2. Getting started with XSLT and XPath (cont'd)

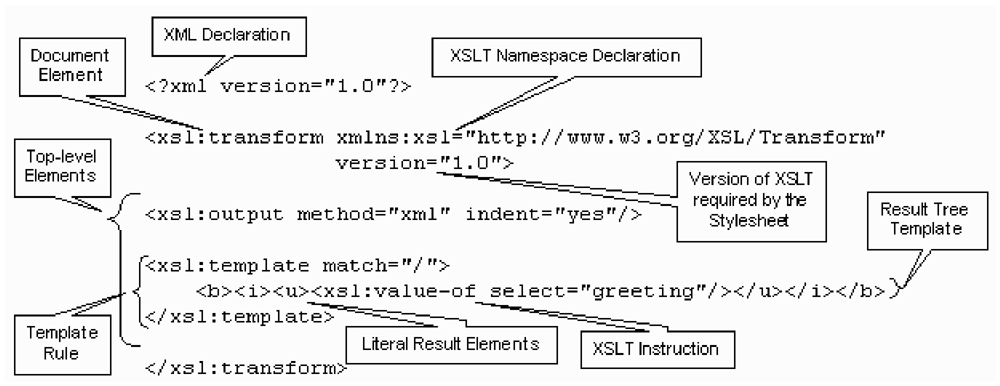
2.2 Syntax basics: Stylesheets, Templates, Instructions

Next we'll look at some basic terminology both helpful in understanding the principles of writing an XSLT stylesheet and recognizing the constructs used therein. This section is not meant as tutelage for writing stylesheets, but only as background information, nomenclature, and practice guidelines.

Note:	I use two pairs of diametric terms not used as such in the XSLT Recommendation itself: explicit/implicit stylesheets and push/pull design approaches. Students of my instructor-led courses have found these distinctions helpful even though they are not official terms. Though these terms are documented here with apparent official status, such status is not meant to be conferred.

2.2.1 Explicitly declared stylesheets

An explicitly declared XSLT stylesheet is comprised of a distinct wrapper element containing the stylesheet specification. This wrapper element must be an XSLT instruction named either `stylesheet` or `transform`, thus it must be qualified by the prefix associated with the XSLT namespace URI. This wrapper element is the document element in a standalone stylesheet, but may in other cases be embedded inside an XML document.



```
<?xml version="1.0"?>
<xsl:transform xmlns:xsl="http://www.w3.org/XSL/Transform"
               version="1.0">
<xsl:output method="xml" indent="yes"/>

<xsl:template match="/">
    <b><i><u><xsl:value-of select="greeting"/></u></i></b>
</xsl:template>

</xsl:transform>
```

Figure 2-4: Components of an Explicit Stylesheet

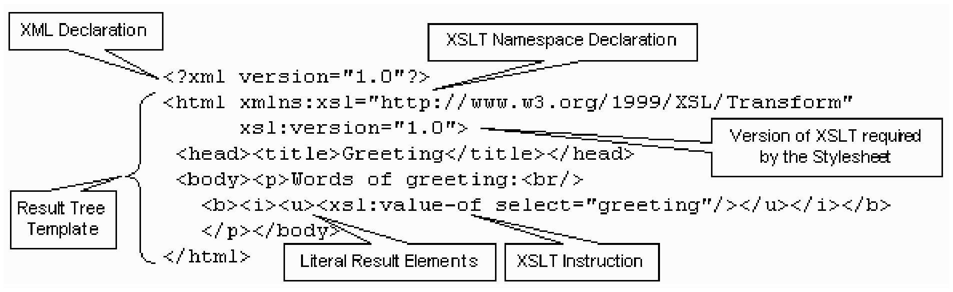
The XML declaration is consumed by the XML processor embedded within the XSLT processor, thus the XSLT processor never sees it. The wrapper element must include the XSLT namespace and version declarations for the element to be recognized as an instruction.

The children of the wrapper element are the top-level elements, comprised of global constructs, serialization information, and certain maintenance instructions. Template rules supply the stylesheet behavior for matching source tree conditions. The content of a template rule is a result tree template containing both literal result elements and XSLT instructions.

The example above has only a single template rule, that being for the root of the document.

2.2.2 Implicitly declared stylesheets

The simplest kind of XSLT stylesheet is an XML file implicitly representing the entire outcome of transformation. The result vocabulary is arbitrary, and the stylesheet tree forms the template used by the XSLT processor to build the result tree. If no XSLT or extension instructions are found therein, the stylesheet tree becomes the result tree. If instructions are present, the processor replaces the instructions with the outcomes of their execution.



```
<?xml version="1.0"?>
<html xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xsl:version="1.0">
 <head><title>Greeting</title></head>
 <body><p>Words of greeting:<br/>
   <b><i><u><xsl:value-of select="greeting"/></u></i></b>
   </p></body>
</html>
```

*Figure 2-5:*Components of an Implicit Stylesheet

The XML declaration is consumed by the XML processor embedded within the XSLT processor, thus the XSLT processor never sees it. The remainder of the file is considered the result tree template for an implicit rule for the root of the document, describing the shape of the entire outcome of the transformation.

The document element is named "`html`" and contains the namespace and version declarations of the XSLT language. Any element type within the result tree template that is qualified by the prefix assigned to the XSLT namespace URI is recognized as an XSLT instruction. No extension instruction namespaces are declared, thus all other element types in the instance are literal result elements. Indeed, the document element is a literal result element as it, too, is not an instruction.

2.2.3 Stylesheet requirements

Every XSLT stylesheet must identify the namespace prefix used therein for XSLT instructions. The default namespace cannot be used for this purpose. The namespace URI associated with the prefix must be the value `http://www.w3.org/1999/XSL/Transform` . It is a common practice to use the prefix `xsl` to identify the XSLT vocabulary, though this is only convention and any valid prefix can be used.

XSLT processor extensions are outside the scope of the XSLT vocabulary, so other URI values must be used to identify extensions.

The stylesheet must also declare the version of XSLT required by the instructions used therein. The attribute is named `version` and must accompany the namespace declaration in the wrapper element instruction as `version="`*`version-number`*`"` . In an implicit stylesheet where the XSLT namespace is declared in an element that is not an XSLT instruction, the namespace-qualified attribute declaration must be used as *`prefix`*`:version="`*`version-number`*`"` .

The version number is a numeric floating-point value representing the latest version of XSLT defining the instructions used in the stylesheet. It need not declare the most capable version supported by the XSLT processor.

2.2.4 Instructions and literal result elements

XSLT instructions are only detected in the stylesheet tree and are not detected in the source tree. Instructions are specified using the namespace prefix associated with the XSLT namespace URI. The XSLT Recommendation describes the behavior of the XSLT processor for each of the instructions defined based on the instruction's element type (name).

Top-level instructions are considered and/or executed by the XSLT processor before processing begins on the source information. For better performance reasons, a processor may choose to not consider a top-level instruction until there is need within the stylesheet to use it. All other instructions are found somewhere in a result tree template and are not executed until that point at which the processor is asked to add the instruction to the result tree. Instructions themselves are never added to the result tree.

Some XSLT instructions are control constructs used by the processor to manage our stylesheets. The wrapper and top-level elements declare our globally scoped constructs. Procedural and process-control constructs give us the ability to selectively add only portions of templates to the result, rather than always adding an entire template. Logically-oriented constructs give us facilities to share the use of values and declarations within our own stylesheet files. Physically-oriented constructs give us the power to share entire stylesheet fragments.

Other XSLT instructions are result tree value placeholders. We declare how a value is calculated by the processor, or obtained from a source tree, or both calculated by the processor from a value from a source tree. The value calculation is triggered when the XSLT processor is about to add the instruction to the result tree. The outcome of the calculation (which may be nothing) is added to the result tree.

All other instructions engage customized non-standard behaviors and are specified using extension elements in a standardized fashion. These elements use namespace prefixes declared by our stylesheets to be instruction prefixes. Extension instructions may be either control constructs or result tree value placeholders.

Consider the simple example in our stylesheets used earlier in this chapter where the following instruction is used:

```
<xsl:value-of select="greeting"/>
```

Example 2-9: Simple value-calculation instruction in Example 2-4

This instruction uses the `select=` attribute to specify the XPath expression of some value to be calculated and added to the result tree. When the expression is a location in the source tree, as is this example, the value returned is the value of the first location identified using the criteria. When that location is an element, the value returned is the concatenation of all of the #PCDATA text contained therein.

This example instruction is executed in the context of the root of the source document being the focus. The child of the root of the document is the document element. The expression requests the value of the child named "`greeting`" of the root of the document, hence, the value of the document element named "`greeting`". For any source document where "`greeting`" is not the document element, the value returned is the empty string. For any source document where it is the document element, as is our example, the value returned is the concatenation of all #PCDATA text in the entire instance.

A literal result element is any element in a stylesheet that is not a top-level element and is not either an XSLT instruction or an extension instruction. A literal result element can use the default namespace or any namespace not declared in the stylesheet to be an instruction namespace.

When the XSLT processor reads the stylesheet and creates the abstract nodes in the stylesheet tree, those nodes that are literal result elements represent the nodes that are added to the result tree. Though the definition of those nodes is dictated by the XML syntax in the stylesheet entity, the syntax used does not necessarily represent the syntax that is serialized from the result tree nodes created from the stylesheet nodes.

Literal result elements marked up in the stylesheet entity may have attributes that are targeted for the XML processor used by the XSLT processor, targeted for the XSLT processor, or targeted for use in the result tree. Some attributes are consumed and acted upon as the stylesheet file is processed to build the stylesheet tree, while the others remain in the stylesheet tree for later use. Those literal result attributes remaining in the stylesheet tree that are qualified with an instruction namespace are acted on when they are asked to be added to the result tree.

2.2.5 Templates and template rules

Many XSLT instructions are container elements. The collection of literal result elements and other instructions being contained therein comprises the XSLT template for that instruction. A template can contain only literal result elements, only instruction elements, or a mixture of both. The behavior of the stylesheet can ask that a template be added to the result tree, at which point the nodes for literal result elements are added and the nodes for instructions are executed.

Consider again the simple example in our stylesheets used earlier in this chapter where the following template is used:

```
<b><i><u><xsl:value-of select="greeting"/></u></i></b>
```

Example 2-10: Simple template in Example 2-4

This template contains a mixture of literal result elements and an instruction element. When the XSLT processor adds this template to the result tree, the nodes for the `<b>` , `<i>` and `<u>` elements are simply added to the tree, while the node for the `xsl:value-of` instruction triggers the processor to add the outcome of instruction execution to the tree.

A template rule is a declaration to the XSLT processor of a template to be added to the result tree when certain conditions are met by source locations visited by the processor. Template rules are either top-level elements explicitly written in the stylesheet or built-in templates assumed by the processor and implicitly available in all stylesheets.

The criteria for adding a written template rule's template to the result tree are specified in a number of attributes, one of which must be the `match=` attribute. This attribute is an XPath pattern expression, which is a subset of XPath expressions in general. The pattern expression describes preconditions of source tree nodes. The stylesheet writer is responsible for writing the preconditions and other attribute values in such a way as to unambiguously provide a single written or built-in template for each of the anticipated source tree conditions.

In an implicitly declared stylesheet, the entire file is considered the template for the template rule for the root of the document. This template rule overrides the built-in rule implicitly available in the XSLT processor.

Back to the simple example in our explicitly declared stylesheet used earlier in this chapter, the following template rule is declared:

```
<xsl:template match="/">
    <b><i><u><xsl:value-of select="greeting"/></u></i></b>
</xsl:template>
```

Example 2-11: Simple template rule in Example 2-4

This template rule defines the template to be added to the result tree when the root of the document is visited. This written rule overrides the built-in rule implicitly available in the XSLT processor. The template is the same template we were discussing earlier: a set of result tree nodes and an instruction.

The XSLT processor begins processing by visiting the root of the document. This gives control to the stylesheet writer. Either the supplied template rule or built-in template rule for the root of the document is processed, based on what the writer has declared in the stylesheet. The writer is in complete control at this early stage and all XSLT processor behavior is dictated what the writer asks to be calculated and where the writer asks the XSLT processor to visit.

2.2.6 Approaches to stylesheet design

The last discussion in this two-chapter introduction regards how to approach using templates and instructions when writing a stylesheet. Two distinct approaches can be characterized. Choosing which approach to use when depends on your own preferences, the nature of the source information, and the nature of the desired result.

Note:	I refer to these two approaches as either stylesheet-driven or data-driven, though the former might be misconstrued. Of course all results are stylesheet-driven because the stylesheet dictates what to do, so the use of the term involves some nuance. By *stylesheet-driven* I mean that the order of the result is a result of the stylesheet tree having explicitly instructed the adding of information to the result tree. By *data-driven* I mean that the order of the result is a result of the source tree ordering having dictated the adding of information to the result tree.

2.2.6.1 Pulling the input data

When the stylesheet writer knows the location of and order of data found in the source tree, and the writer wants to add to the result a value from or collection of that data, then information can be pulled from the source tree on demand. Two instructions are provided for this purpose: one for obtaining or calculating a single string value to add to the result; and one for adding rich markup to the result based on obtaining as many values as may exist in the tree.

The writer uses the `<xsl:value-of select="`*`XPath-expression`*`"/>` instruction in a stylesheet's element content to calculate a single value to be added to the result tree. The instruction is always empty and therefore does not contain a template. This value calculated can be the result of function execution, the value of a variable, or the value of a node selected from the source tree. When used in the template of various XSLT instructions the outcome becomes part of the value of a result element, attribute, comment, or processing instruction.

Note there is also a shorthand notation called an "attribute value template" that allows the equivalent to `<xsl:value-of>` to be used in a stylesheet's attribute content.

To iterate over locations in the source tree, the `<xsl:for-each select="`*`XPath-node-set-expression`*`">` instruction defines a template to be processed for each instance, possibly repeated, of the selected locations. This template can contain literal result elements or any instruction to be executed. When processing the given template, the focus of the processor's view of the source tree shifts to the location being visited, thus providing for relative addressing while moving through the information.

These instructions give the writer control over the order of information in the result. The data is being pulled from the source on demand and added to the result tree in the stylesheet-determined order. When collections of nodes are iterated, the nodes are visited in document order. This implements a stylesheet-driven approach to creating the result.

An implicitly-declared stylesheet is obliged to use only these "pull" instructions and must dictate the order of the result with the above instructions in the lone template.

2.2.6.2 Pushing the input data

The stylesheet writer may not know the order of the data found in the source tree, or may want to have the source tree dictate the ordering of content of the result tree. In these situations, the writer instructs the XSLT processor to visit source tree nodes and to apply to the result the templates associated with the nodes that are visited.

The `<xsl:apply-templates select="`*`XPath-node-expression`*`">` instruction visits the source tree nodes described by the node expression in the `select=` attribute. The writer can choose any relative, absolute, or arbitrary location or locations to be visited.

Each node visited is pushed through the stylesheet to be caught by template rules. Template rules specify the template to be processed and added to the result tree. The template added is dictated by the template rule matched for the node being pushed, not by a template supplied by the instruction when a node is being pulled. This distinguishes the behavior as being a data-driven approach to creating the result, in that the source determines the ultimate order of the result.

An implicitly-declared stylesheet can only push information through built-in template rules, which is of limited value. As well, the built-in rules can be mimicked entirely by using pull constructs, thus they need never be used. There is no room in the stylesheet to declare template rules in an implicitly-declared stylesheet since there is no wrapper stylesheet instruction.

An explicitly-declared stylesheet can either push or pull information because there is room in the stylesheet to define the top-level elements, including any number of template rules required for the transformation.

Putting it all together

We are *not* obliged to use only one approach when we write our stylesheets. It is very appropriate to push where the order is dictated by the source information and to pull when responding to a push where the order is known by the stylesheet. The most common use of this combination in a template is localized pull access to values that are relative to the focus being matched by nodes being pushed.

Note that push-oriented stylesheets more easily accommodate changes to the data and are more easily exploited by others who wish to reuse the stylesheets we write. The more granularity we have in our template rules, the more flexibly our stylesheets can respond to changes in the order of data. The more we pull data from our source tree, the more dependent we are on how we have coded the access to the information. The more we push data through our stylesheet, the less that changes in our data impact our stylesheet code.

Look again at the examples discussed earlier in this article and analyze the use of the above pull and push constructs to meet the objectives of the transformations.

These introductions and samples in this article have set the context, and only scratch the surface of the power of XSLT to effect the transformations we need when working with our structured information.

XML.com has continuing coverage and tutorials about XPath and XSLT in its regular column, Transforming XML.

This is a prose version of an excerpt from the book "Practical Transformation Using XSLT and XPath" (Eighth Edition ISBN 1-894049-05-5 at the time of this writing) published by Crane Softwrights Ltd., written by G. Ken Holman; this excerpt was edited by Stan Swaren, and reviewed by Dave Pawson.